Programming in VAX-11

digital software

Programming in
VAX-11 C

May 1982

This manual defines the VAX-11 C programming language and provides the information necessary for developing C programs on VAX-11 computers.

Programming in VAX-11 C

AA-L370A-TE

Software Version V1.0

digital equipment corporation · maynard, massachusetts

First Printing, May 1982

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DECsystem-10	PDT
DECUS	DECSYSTEM-20	RSTS
DIGITAL	DECwriter	RSX
PDP	DIBOL	VMS
UNIBUS	EduSystem	VT
VAX	IAS	
DECnet	MASSBUS	digital

ZK2164

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

Contents

Examples

Figures

Tables

Preface

Scope of This Manual

This manual combines reference information on the VAX–11 C language with the information necessary for developing and debugging C programs on a VAX/VMS system.

Who Can Use This Manual

Most readers of this manual should be familiar with at least one other high-level programming language, possibly several. Therefore, the manual does not try to teach VAX–11 C to new programmers. Instead, it answers the questions that are usually posed by experienced programmers who want to learn a new language quickly.

Where to Find More Information

If you would like a more tutorial introduction to the C language, see *The C Programming Language*,[1] by the designers of the language. That book contains extensive examples of programs written in C and can serve as an introduction to the C language in particular or to programming in general. Note that VAX–11 C contains features and enhancements to the C programming language. Therefore, *Programming in VAX–11 C* should be used as *the* reference book for the full description of VAX–11 C.

Unless you do all your programming in C and make little or no use of features that depend on VAX/VMS or the VAX–11 machine architecture, you should have all or most of the VAX/VMS system documentation available for reference. For a complete list of all VAX/VMS documents and their order numbers, see the *VAX–11 Information Directory and Index*.

---

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice-Hall, 1978).

Conventions Used in This Document

Enter string>Abcd RET	In computer dialogs, the user's response to a prompt is printed in red.
float x; . . x = 5 ;	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
option,...	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma that precedes the ellipsis indicates that successive items must be separated by commas.
quotation mark apostrophes	The term quotation mark is used to refer to the quotation mark symbol ("). The term apostrophe is used to refer to the single quotation mark symbol (´).
switch statement **fprintf** function	Boldface type identifies language keywords and the names of library functions.
[output-source,...]	Square brackets, in function synopses and a few other contexts, indicate that a syntactic element is optional.
sc-specifier ::= **auto** **static** **extern** **register** **typedef**	In syntax definitions, items appearing on separate lines are mutually exclusive alternatives.
declarator *initializer*	Italics, in syntax definitions and most other contexts, indicate that a syntactic element is optional.
△	A delta symbol is used in some contexts to indicate a single ASCII space character.
RET	The symbol RET represents a single stroke of the RETURN key on a terminal.
CTRL/X	The symbol CTRL/X, where x is a letter, represents a terminal control character, generated by holding down the CTRL key while pressing the letter key.

Chapter 1

A Brief Discussion of C

Because most readers of this manual are familiar with at least one other high-level programming language, the manual does not try to teach VAX-11 C to new programmers. Instead, it answers the questions typically posed by experienced programmers who want to learn a new language quickly. These questions include:

- What are the language's data types?
- What operations can be performed on particular types? What conversion and type checking take place when different types are mixed in an operation?
- How is the flow of a program controlled? That is, by what means are such fundamental tools as decisions, loops, and functions written in the language?
- What is the structure of a program?
- To what extent is the language standardized, in the practical sense that a program compiled on one manufacturer's system can also be compiled, with a minimum of change, on another system?
- What distinguishes this language from others?

These questions are addressed briefly in this chapter, and in more detail later.

If you are already familiar with the C programming language, you will find that VAX-11 C is fundamentally the same language as that to which you are accustomed. You might want to start with Chapter 12, Program Development. That chapter and those that follow it explain how to develop C programs on a VAX/VMS system.

Even if you are already familiar with C on other implementations, you would be well advised to consult Chapters 1 through 11 to note the extensions and relaxed rules specific to VAX-11 C.

1.1 Data Types

Programming languages seem to fall into two groups with respect to the way they represent data. Some, like PL/I, use forms that are closely related to particular applications. Consequently, such languages have

many data types and, of necessity, have complex rules for operations on various types and conversions between types. Other languages, C among them, use data types that are closely related to the way computers are built. C has a small set of fundamental types in which scalar variables can be declared as:

- Integers of various, fixed sizes. Integers can be signed or, if declared with the keyword **unsigned**, unsigned. The integer keywords (in VAX–specific sizes) are:
  - — **char**, an 8-bit byte, usually used to represent an ASCII character
  - — **short**, or **short int**, a 16-bit integer
  - — **int**, or **long int**, a 32-bit integer
- Floating-point numbers, in two fixed sizes:
  - — **float**, a single-precision floating-point number (represented in the VAX–11 F floating-point format)
  - — **double**, a double-precision floating-point number (represented in the VAX–11 D floating-point format)
- **enum** values, which are scalars of a user-defined, or enumerated, type. As in PASCAL, you can define new scalar types in C by writing a type name followed by an ordered list of identifiers that are the constants of that type.
- Pointers. On the VAX–11, pointer variables contain 32-bit addresses of other variables.

It can be seen from this list that C's scalar types were chosen for ease of implementation, since most machines can represent integers and floating-point numbers directly. In most implementations, VAX–11 C included, **enum** values are represented in some internal integer format. However, in programs they should be treated as having a type distinct from the arithmetic types.

C compilers differ in the exact size of a particular kind of variable, but all use machine representations that are natural to the machine in question. For example, VAX–11 C uses a longword to represent the default size integer (**int**). The language provides a standard operator, **sizeof**, which yields the size of its operand in bytes, where a byte is the amount of storage required to hold one character in the machine's native character set. Therefore, in any implementation, **sizeof(char)** is usually 1. The **sizeof** operator can be used to determine the size of other data types on a particular machine.

As in virtually all programming languages, you can also represent data in aggregates of associated items. C has three kinds of aggregate:

- The array — an aggregate of items, or elements, which all must have the same data type. Arrays may be multidimensional. Character strings in C are declared as one-dimensional arrays of type **char**, either explicitly in string variables or implicitly in character-string constants.

- The structure — an aggregate of items, or members, which can have the same or different data types. The amount of storage occupied by a structure is the sum of the sizes of all its members.
- The union — an aggregate similar to the structure, but whose storage holds the value of only one member at a time. That is, the storage occupied by a union is as large as its largest member.

1.2 Operations

C has a very large set of operators, as shown in Table 1-1.

Table 1-1: Summary of C Operators

Operator	Example	Result
- [unary]	-a	negative of a
* [unary]	*a	reference to object at address a
& [unary]	&a	address of a
~	~a	one's complement of a
++ [prefix]	++a	a after increment
++ [postfix]	a++	a before increment
-- [prefix]	--a	a after decrement
-- [postfix]	a--	a before decrement
sizeof	sizeof(t1)	size in bytes of type t1
	sizeof e	size in bytes of expression e
(type-name)	(t1)e	expression e, converted to type t1
+	a + b	a plus b
- [binary]	a - b	a minus b
* [binary]	a * b	a times b
/	a / b	a divided by b
%	a % b	remainder of a/b (a modulo b)
>>	a >> b	a, right-shifted b bits
<<	a << b	a, left-shifted b bits
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise
!=	a != b	1 if a not equal to b; 0 otherwise
& [binary]	a & b	bitwise AND of a and b
|	a | b	bitwise OR of a and b
^	a ^ b	bitwise XOR (exclusive OR) of a and b
&&	a && b	logical AND of a and b (yields 0 or 1)
||	a || b	logical OR of a and b (yields 0 or 1)
!	!a	logical NOT of a (yields 0 or 1)

Table 1-1: (Cont.) Summary of C Operators

Operator	Example	Result
?:	a ? e1 : e2	expression e1 if a is nonzero, expression e2 if a is zero
=	a = b	a (with b assigned to a)
+=	a += b	a plus b (assigned to a)
-=	a -= b	a minus b (assigned to a)
*=	a *= b	a times b (assigned to a)
/=	a /= b	a divided by b (assigned to a)
%=	a %= b	remainder of a/b (assigned to a)
>>=	a >>= b	a, right-shifted b bits (assigned to a)
<<=	a <<= b	a, left-shifted b bits (assigned to a)
&=	a &= b	a AND b (assigned to a)
|=	a |= b	a OR b (assigned to a)
^=	a ^= b	a XOR b (assigned to a)
,	e1,e2	e2 (e1 evaluated first)

The operands of bitwise operators must be integral; that is, they must be signed or unsigned integers. The operands of the arithmetic or comparison operators are converted automatically to a common type by a set of rules known as the usual arithmetic conversions (see Section 4.1.1).

The restrictions that do exist on operand types are intended to prevent meaningless or unrepresentable results. For instance, integers and pointers may be used together in some expressions, but not in others where they would probably produce meaningless addresses. Expressions that *can* appear on the left side of an assignment are called lvalues, to distinguish them from other expressions.

Note that it is conventional in C for a logical "true" value to be equivalent to the condition "is not equal to zero." Similarly, "false" is equivalent to "is equal to zero." It is important to realize that a true/false status may be conveyed by other than the integer constants 0 and 1 or the least significant bit of an integral value. In C, the constants 200 and 3.297e15, for example, are both "true".

1.3 Program Control

As in other languages, C has keywords and operators that make decisions, control loops, and call functions.

1.3.1 Decisions and Transfers of Control

Decisions can be made in C programs with the **if** and **else** keywords, and with the conditional operator.

The **if** statement is a compound statement involving a logical expression:

```
if (expression) statement
```

The statement is executed if the expression is true (that is, has a nonzero value). The statement can be either a single statement or a compound statement enclosed in braces ({}). It can be followed by an **else** statement, which provides an alternative statement to be executed if the logical expression is false (zero).

A similar kind of operation (but one that produces a value) can be effected with the conditional operator:

```
e1 ? e2 : e3
```

The conditional operator (?:) means: evaluate expression e1, and then evaluate expression e2 if the result of e1 is nonzero. If the result of e1 is zero, evaluate expression e3 instead.

C has facilities to transfer control both conditionally and unconditionally:

- The **goto** statement transfers control unconditionally to a given label.
- The **switch** statement transfers control to one of a list of cases, depending on the value of a given expression.
- The **break** and **continue** statements perform transfers of control in loop and **switch** statements. **break** terminates a loop or **switch** and transfers control to the next statement; **continue** transfers control to the bottom of the enclosing loop.

1.3.2 Loops

Loops are implemented in C with the **for**, **while**, and **do** statements. The distinctions are as follows:

- The **do** statement executes its body one or more times; a given expression is evaluated after each execution of the body, and if the result is zero the **do** loop terminates.
- The **while** statement is the same as a **do** statement except that the controlling expression is evaluated before each iteration. That is, the **while** body may not be executed at all, whereas the **do** body is always executed at least once.
- The **for** statement has three user-specified expressions and a body. The first expression is executed before the first iteration of the body; usually, it initializes a control variable. The second expression is executed before every iteration, including the first; execution of the loop proceeds only if the result of the second expression

is nonzero. The third expression is executed at the end of every iteration; it usually controls the change made to the control variable. For example:

```
for (i = 0; i < 10; i++) statement
```

executes the statement 10 times, because i < 10 becomes false (zero) after the tenth iteration. In other words, the **for** statement above is equivalent to:

```
i = 0;
while(i < 10)
     {
     statement
     i++;
     }
```

1.3.3 Function Calls

A C function can call any other function simply by declaring its name. Once declared, the name, followed by parentheses, can be used in expressions. For example, the following program segment first declares a function (dfun) and then calls it from a separately compiled source file:

```
/* MAIN (CALLING) FUNCTION */
main()
{
/* DECLARE x AND t AS DOUBLE-PRECISION
 * VALUES, dfun AS A FUNCTION RETURNING
 * SUCH A VALUE
 */
double x,t,dfun();
/* CALL dfun, ASSIGNING RETURNED
 * VALUE TO t
 */
t = dfun(x);
}
```

A function call like the one shown here is performed the same way whether the called function is written in C or in some other VAX-11 native mode language, such as FORTRAN, MACRO, or PL/I.

1.4 Program Structure

A C program is made up of one or more function definitions. For example, a function named lower might be defined as follows:

```
/* CONVERT UPPERCASE TO LOWERCASE */
lower(c_up)
int c_up;
{
```

```
    if (c_up >= 'A' && c_up <= 'Z')
        return (c_up + 'a' - 'A');
    else return (c_up);
}
```

The definition shows, in the first two lines, that lower has one parameter, c_up, and that c_up is an integer. The action taken by the function, or the body, consists of the statements between the braces ({}). The execution of a function terminates when a **return** statement is executed or when the right brace is encountered.

This example also shows a few other features of C program structure:

- C is a free-form language, like PL/I or PASCAL. The meaning of a program is independent of the placement of text on a line or page.
- Programs may be commented freely, with comment text written between the character pairs /* and */ wherever spaces are valid. A comment can have more than one open-comment delimiter (/*), but it can have only one close-comment delimiter (*/). This makes it easy to convert unwanted source lines into comments.

All C functions are external; a function definition may only call other functions, not define them. Therefore, C is not a block-structured language as the term is usually applied. However, C does have facilities for controlling the scope of variables. The scope of a variable can be part of a function, the entire function, several functions in a program, or the entire program.

In addition to function definitions, programs may also contain data definitions that are external to any of the program's functions. An external data definition is one means of extending the scope of a variable beyond a single function, so that information can be shared among the functions in the program.

Keywords in C identify data types (such as **int** and **char**), storage classes (such as **auto** and **static**), and certain statements (such as **if** and **return**). C keywords are predefined identifiers that may not be redeclared by the programmer. Keywords must be lowercase, and they cannot be abbreviated.

Note that many identifiers that have special meaning in C programs are not keywords. For example, C recognizes main to be the name of the main function. If any function in a program is named main, execution starts with that function. However, since it is not a keyword, main can be used in other contexts, too. Similarly, the names of library functions, which in C perform such fundamental operations as input and output, are not keywords.

1.5 Sample Program

If you are not familiar with the C programming language, you may want to compare a common algorithm in C with the same algorithm in a familiar language.

The sample program in this section (Examples 1–1 through 1–3) is a Shell sort algorithm. A Shell sort first compares elements in the unsorted data that are far apart (before comparing adjacent ones), and then it places them in the proper order. The gap between compared elements is initially half the size of the unsorted data array. The gap gradually diminishes until the program is comparing adjacent elements. Thus, the data end up in the correct order.

```
C       Sort array of integers into ascending order.
C       Declare loop counter and array; initialize array.
        INTEGER I,ARRAY(0:9)
        DATA ARRAY/10,9,8,7,6,1,2,3,4,5/
C       Sort the data in the array.
        CALL SHELL(ARRAY,10)
C       Write out the sorted data.
        DO 10 I = 0,9
10      WRITE(6,15) ARRAY(I)
15      FORMAT(' ',I2)
        END
        SUBROUTINE SHELL(V,N)
C       To make these comparisons easier, begin subscripts
C       at zero.
        INTEGER V(0:N-1)
        INTEGER GAP,I,J,TEMP
C       Initialize the gap to half the array size.
        GAP = N/2
C       Perform loop from statements 10 to 99 until the
C       gap is zero.
10      IF (GAP .LE. 0) GO TO 100
C       Now compare the elements of each pair that is
C       separated by the gap, and reverse any that are
C       out of order.
         I = GAP
20       IF (I .GE. N) GO TO 90
          J = I-GAP
30        IF ( (J .LT. 0) .OR. (V(J) .LE. V(J+GAP)) ) GO TO 80
C       Values are out of order: reverse them.
            TEMP = V(J)
            V(J) = V(J+GAP)
            V(J+GAP) = TEMP
            J = J-GAP
            GO TO 30
80        CONTINUE
         I = I+1
         GO TO 20
90       CONTINUE
C       Reduce the gap
        GAP = GAP/2
99      GO TO 10
100     RETURN
        END
```

Example 1-1: Shell Sort in FORTRAN

```
program main(output);

const
    MAXSUB = 9;   (* MAXIMUM SUBSCRIPT *)

    (* DECLARE THE TYPE OF ARGUMENT
    TO BE PASSED TO THE SHELL PROCEDURE *)

type
    arrtype = array[0..MAXSUB] of integer;

var
    i: integer;
    arr: arrtype;  (* arr IS AN ARRAY OF INTEGERS *)

value
    arr := (10,9,8,7,6,1,2,3,4,5); (* INITIALIZE ARRAY *)

     (* PASS THE ARRAY TO shell BY REFERENCE *)

procedure shell (var v: arrtype; n: integer);

    var
      gap,i,j,temp: integer;
    begin
    gap := n div 2;
    while (gap > 0) do
      begin
      i := gap;
      while (i < n) do
        begin
        j := i-gap;
        while ( (j >= 0) and (v[j] > v[j+gap]) ) do
          begin
          temp := v[j];
          v[j] := v[j+gap];
          v[j+gap] := temp;
          j := j-gap
          end;
        i := i+1;
        end;
      gap := gap div 2
      end
    end;
begin (* MAIN PROCEDURE *)

shell(arr,10);
for i := 0 to MAXSUB do writeln(arr[i])

end (* END MAIN PROCEDURE *)
.    (* END OF PROGRAM *)
```

Example 1-2: Shell Sort in PASCAL

```
main() /* SORT ARRAY OF INTEGERS INTO ASCENDING ORDER */     ❶
{                                                           ❷
     static int array[] = {10,9,8,7,6,1,2,3,4,5};           ❸
     int i;

     shell(array,10);/* PASS THE ARRAY TO shell */          ❹

     /* WRITE OUT THE SORTED DATA */
     for (i=0; i<10; i++)                                   ❺
          printf("%d\n",array[i]);
}                                                           ❻

shell(v,n)                                                  ❼
int v[],n;                                                  ❽
{
     int gap,i,j,temp;

     for (gap = n/2; gap > 0; gap /= 2)                     ❾
       for (i = gap; i < n; i++)
         for (j = i-gap; j>=0 && v[j]>v[j+gap]; j -= gap)
              {
              temp = v[j];                                  ❿
              v[j] = v[j+gap];
              v[j+gap] = temp;
              }
}
```

Example 1-3: Shell Sort in C

The following notes are keyed to the circled numbers in Example 1-3:

❶ The definition of the main function and execution of the program begin here. The empty parentheses (required syntax) indicate that the main function in this program has no parameters.

❷ The left brace ({) begins a compound statement. This character is required; there are no substitutes, such as brackets or parentheses. Compound statements can be used wherever single statements are valid, such as in the bodies of loops.

❸ This line illustrates the declaration of a variable — here, a static array of integers. Ten initializers are supplied (in the braces). The size of the array, normally an integer within the brackets ([]), is omitted in this case, and the size is determined automatically by the number of initializers. The size of an array in C is the number of its elements, not the maximum subscript. The size in this example is 10, so the subscripts range from 0 to 9.

Note also that the line is indented. Indentation is not semantically meaningful in C, but it is used to show subordination. The statements in the body of the main function are usually indented under the braces that enclose the body.

❹ This line illustrates a function call in C; here, the shell function is called. The format of Example 1–3 suggests that main and shell are in the same file, but they could be in separately compiled files, too.

Notice that shell has not been formally declared in the main function. The C default in such situations is to assume that shell is a function returning an integer. Even when declarations are present in the calling function, they show only the function's name and the type of its return value, not the number or types of the parameters.

The identifier array and the constant 10 are shell's arguments. The number and types of arguments in a function call must exactly match the declaration of the external function's parameters.

❺ The **for** statement is one of C's facilities for controlling program loops. The body of a **for** statement is a single or compound statement, which follows the **for** statement and a list of expressions. Here, the body is a call to the function **printf**, which is executed 10 times to write out the sorted values in the array.

❻ The right brace ends a compound statement, which in this case is the body of the main function. The execution of functions ends either when a **return** statement is executed or when the terminating right brace is encountered.

❼ This line begins the definition of the function shell, which has two parameters.[1] The parameters need not have the same identifiers as the arguments in the calling function.

❽ This line declares the parameters v[ ] and n. Parameter declarations precede the function body. They show the number and types of the function parameters — in this case "array of integers" and "integer". The function arguments must match the parameter definitions in both number and type.

❾ This line begins a series of nested **for** loops. Notice that, as a result of C's grammar, all three **for** statements precede the body of the innermost loop.

❿ This line begins the **for** body, which performs the actual exchange of values in the array. Notice that the body is also the syntactic end of the shell function; that is, shell does not have a **return** statement.

---

1. The term *parameter* in VAX-11 C is comparable to *parameter* in PL/I or *dummy argument* in FORTRAN; it denotes a variable declared in the called function, as opposed to *argument*, which denotes the expression written in the function call.

The sorted array elements are returned to the main function through the parameter v. Because v's corresponding argument is the address of the first element of the main function's array, references to v's elements are also references to the elements of array. The bracket operator (used for array subscripting) is actually performing address arithmetic to determine the appropriate element of the main function's array. (The bracket operator and address arithmetic are explained in Chapter 4.)

However, except in cases of implicit or explicit address passing, you should regard C as a language that passes arguments by value. That is, an argument's value is calculated and copied into the corresponding parameter.

1.6 Degree of Standardization

There is no ANSI standard or other industry-wide standard for the C programming language. The C language is described in the book written by the designers of C,[1] and in a series of technical notices published by the American Telephone and Telegraph Company.

Certain features are fundamental to C and exist in most C compilers, including the VAX-11 C compiler. These features are described in this manual as follows:

- All basic elements of program structure are documented in Chapter 2. (Chapter 2 points out that VAX-11 C allows certain nonportable characters in identifiers and also has the nonportable storage classes **globaldef**, **globalref**, **globalvalue**, and **readonly**.)
- The data types and declaration rules are described in Chapter 3.
- The set of (and rules governing) operators are described in Chapter 4. These rules describe the format of a function call, although they do not describe a set of available functions. The C language has no built-in or otherwise predefined functions.
- The set and semantics of statements are documented in Chapter 5.

In addition to the compiler, virtually every C language implementation includes: (1) a preprocessor facility; (2) a set of definition files (conventionally identified by the file type H); and (3) a library of run-time functions. Compared with the features listed above, these aspects of a C implementation are more loosely standardized and depend on the particular machine for which the C implementation was designed. The following commentary suggests guidelines for using these features to write portable programs:

---

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice-Hall, 1978).

1. In most respects, the preprocessor control lines described in Chapter 7 are compatible with their equivalents in other implementations. Note, however, that VAX-11 C allows constant expressions of any integral type in an **#if** preprocessor control line. These are valid C expressions, evaluated by the usual rules, that have constant results. (In this context, constant expressions cannot include the **sizeof** operator.) This set of expressions may be more extensive than some compilers allow. A reasonable precaution is to limit **#if** expressions to actual constants.

   In addition, some aspects of the macro substitutions performed by the VAX-11 C **#define** control line may differ from other implementations. Because this feature is so widely used in C programming, the best precaution is to compare the results of the different compilers.

2. The VAX-11 C definition files are designed to be semantically compatible with those of other known implementations. The text in each file may not be identical with other manufacturers' versions, but in most cases, you can expect a program that uses such files to compile successfully and to function correctly.

   In addition, the search order for **#include** files is probably unique to the VAX-11 implementation, although it is designed to be useful and functionally similar to other implementations. The main work for you may be to place the included file in the appropriate directory. See Chapter 7.

3. The functions in the VAX-11 C run-time library are documented in Chapter 6. There is no guarantee that run-time functions are absolutely compatible with those of other implementations. However, the input/output functions, character and string functions, mathematical functions, and memory allocation functions are common features, and these functions in VAX-11 C are compatible with those of other implementations. It is possible that the set of miscellaneous functions may be specific to the VAX-11 and UNIX[1] implementations. The UNIX emulation functions are included in VAX-11 C for the specific purpose of emulating UNIX system functions; they may not occur in most C implementations.

4. Finally, the following functions are specific to VAX-11 C:

   - **strspn** and **strcspn**
   - **strpbrk**
   - **creat**, in the form that includes RMS keywords
   - **delete**

1. UNIX is a trademark of Bell Laboratories.

Transporting programs between VAX-11 C and other C implementations may require minor program modifications. VAX-11 C provides the following predefined constants for constructing portable programs; all are defined to be nonzero ("true" in C's logic tests):

```
vax
vms
vax11c
```

These symbols can be used, for example, in **#if** and **#ifdef** control lines to determine whether source code that may not be portable should be compiled. Control lines are described in Chapter 7.

Appendix A contains more information on compatibility between VAX-11 C and other C compilers. The information in Appendix A may be useful when transporting C programs.

The VAX-11 C compiler command, CC, also has a /STANDARD=PORTABLE qualifier that detects most nonportable constructions and issues appropriate warning messages.

Chapter 2

Program Structure

This chapter describes the two basic elements of a C program: function definitions and external data definitions.

Functions define the actions performed by a program. Section 2.1 describes C function definitions and explains the following associated elements of the C language:

- Function names
- Function parameters and arguments
- Identifiers
- Blocks
- Comments
- Keywords

External data definitions provide an alternative to function parameters for exchanging information among several functions in a program. They are described in Section 2.2.

2.1 Function Definitions

Example 2–1 shows a simple C program that consists of two function definitions; the components of a function definition are labeled in the second function.

The name lower begins a new function definition; the function lower has a single parameter, c_up. (Notice that although main has no parameters, the pair of parentheses must be present.)

The next statement, int c_up, declares the parameter's data type — in this case, **int** for integer. The declaration is omitted if the function has no parameters; furthermore, declarations in this place in the program may specify only the names of parameters, not the names of other variables.

```
#include stdio

/* PROGRAM THAT CONVERTS ITS INPUT TO LOWERCASE */
main()
{
     FILE *infile, *outfile;
     int i,c,c_out;

     /* OPEN INFILE FOR INPUT */
     infile = fopen("ex113.in","r");

     /* OPEN OUTFILE FOR OUTPUT */
     outfile = fopen("ex113.out","w");

     while ((c = getc(infile)) != EOF)
          {
          c_out = lower(c);
          putc(c_out,outfile);
          }
}

/* BEGINNING OF FUNCTION DEFINITION,
 * GIVING FUNCTION NAME AND PARAMETER NAME
 */
lower(c_up)

/* DECLARATION OF PARAMETER TYPE */
int c_up;

/* BEGINNING OF FUNCTION BODY */
{
     if (c_up >= 'A' && c_up <= 'Z')
             return (c_up + 'a' - 'A');
     else return (c_up);

/* END OF FUNCTION BODY;
 * END OF FUNCTION DEFINITION
 */
}
```

Example 2-1: Case Conversion Program

The left brace ({) signifies the beginning of the function body; a right brace (}) signifies the end. The function body is any set of valid C statements. Usually, the body includes one or more **return** statements, as shown here. A **return** statement can specify an expression whose value is returned to the calling function. If the expression is omitted, the returned value is undefined in the calling function. If the **return** statement is not included, the function terminates when the right brace is encountered, and its return value is undefined.

2.1.1 Main Function and Function Names

The execution of a program begins at a function named main, or, if there is no function with this name, at the first function seen by the VAX/VMS Linker. The word *main* is not a language keyword, however, so it may be used for other purposes in the program. In Example 2–1, the main function physically precedes the function lower, but the two function definitions could appear in the other order.

Function names have compile-time scoping rules that are slightly different from those that apply to other identifiers. Any valid function identifier followed by a left parenthesis is declared implicitly as the name of a function returning **int.** In Example 2–1, the name lower is not declared in the main function even though it is used there. If lower were to return something other than an integer, its name would have to be declared in the main function with the type of the returned value. For example:

```
main()
{
/* DECLARE lower AS FUNCTION RETURNING A CHARACTER */
     char lower();

     FILE *infile, *outfile;
     int i,c,c_out;

     infile = fopen("ex113.in","r");
     outfile = fopen("ex113.out","w");

     while ((c = getc(infile)) != EOF)
          {
          c_out = lower(c);
          putc(c_out,outfile);
          }
}

char lower(c_up)
int c_up;
{
.
.
}
```

A function name can also be used without parentheses and arguments. It is then treated as the address of the function of that name. However, functions must be defined or declared before they can be referenced in this way. A typical use is in a list of arguments, to pass the address of a function to another function as one of the arguments.

In the following example, the main function references two functions, x and y. The function x is defined before the main function, so it is not declared in the main function. The function y must be declared because its definition follows the main function. The statement funct(x,y) passes the addresses of the two functions to the function funct, which is contained in a separately compiled source file.

```
/* x() IS DEFINED BEFORE IT IS USED */
x() { return 25; }

main()
{
/* y() MUST BE DECLARED BEFORE IT IS USED */
      int y();
      .
      .
/* x AND y ARE PASSED AS THE ADDRESSES
   OF THE FUNCTIONS */
      funct(x,y);
      .
      .
}
y() { return 30; }

In a separate compilation:

funct(f1,f2)
/* DECLARES THE ARGUMENTS AS POINTERS TO FUNCTIONS
 * RETURNING INTEGERS
 */
int (*f1)(), (*f2)();
{
      .
      .
}
```

Between the definition of a function's name and the declaration of its parameters, you can write the option:

```
main_program
```

This option identifies the function as the main function in the program. It is not a keyword, and it can be spelled in either upper- or lowercase. Use it when the program does not contain a function named main and when you do not want the program's execution to begin at the first

function linked. For example, the following definition establishes the function lower as the main function; execution begins there, regardless of the order in which the function is linked:

```
char lower(c_up)
MAIN_PROGRAM
int c_up;
{
.
.
}
```

NOTE

The main_program option is a VAX-11 C language extension.

2.1.2 Parameters and Arguments

C functions can exchange information by means of parameters and arguments. In this manual, the term *parameter* denotes the variable (in parenthesis) named in a function definition; the term *argument* denotes an expression that is part of a function call. In Example 2–1, the function lower has a single parameter, c_up. When this function is called from main, the argument c is evaluated and passed to lower.

The following rules apply to parameters and arguments of C functions:

- The number of arguments in a function call must always be the same as the number of parameters in the function definition. This number may be zero.
- In VAX-11 C, the maximum number of arguments (and corresponding parameters) is 253 for a single function.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments may be evaluated by the compiler in any order. (In other words, you should not expect function calls or other complicated expressions in the argument list to be evaluated in any particular order.)
- In C, all arguments are passed by value, that is, when a function is called, the parameter receives a copy of the argument's value, not its address. The rule applies to all scalar variables and to structures and unions passed as arguments. Strictly speaking, a function cannot modify the values of its arguments. Of course, since arguments can be addresses or pointers, a function can use addresses to modify the values of variables defined in the calling function.
- The types of evaluated arguments must match the types of their corresponding parameters. When a function is called, C does not compare the types of the arguments with those of the correspond-

ing parameters and thus does not generally convert the arguments to the types of the parameters. Instead, all of the expressions in the argument list are converted according to the following conventions:

— Any arguments of type **float** are converted to **double**.

— Any arguments of types **char** or **short** are converted to **int**.

— Any arguments of types **unsigned char** or **unsigned short** are converted to **unsigned int**.

— Any function name appearing as an argument is converted to the address of the named function. The corresponding parameter must be declared as a pointer to a function, where the function returns a value of the same type as the function named as an argument.

— Any array name appearing as an argument is converted to the address of the first element of the array. (An array name is the identifier used to declare an array, either without the pair of brackets that usually enclose a subscript, or with fewer pairs of brackets than appear in the array's declaration.) The corresponding parameter can be declared either as an array of the given type or as a pointer to the given type. Since character-string constants are declared implicitly as arrays of characters, this rule also applies to the use of string constants as arguments.

No other conversions are performed on arguments. If you know that a particular argument must be converted to match the type of the corresponding parameter, use the cast operator, described in Chapter 4.

2.1.3 Identifiers

Identifiers can consist of letters, digits, dollar signs ($), and the underscore (_). The first character must not be a digit. An identifier can contain any number of characters, but its first 31 characters must be unique.

The dollar sign should be used only in identifiers for VAX/VMS global symbols. Identifiers that contain dollar signs may not be portable.

Upper- and lowercase letters specify different identifiers. That is, abc and ABC are interpreted as different names by the compiler. You must spell language keywords in lowercase. (Note that the debugger converts all lowercase identifiers to uppercase, and identifiers that are unique to the compiler, such as abc and ABC, will not be unique to the debugger. See Chapter 15 for more information on the debugger.)

Use the following conventions if practical:

- Spell identifiers in uppercase if they are constants that are given values by the **#define** control line.

- Spell all instances of a global name (for example, a name declared with **globalvalue**) in the same case. All names that become part of the VAX/VMS Linker's global symbol table are represented there in uppercase. For example, the compiler would consider

```
int globalvalue ss$_accvio = 0;
globalvalue SS$_ACCVIO;
```

  to denote different global names; however, uppercase forms for both are passed to the linker, potentially causing errors when the program is linked or executed.

- Spell all other identifiers and keywords in lowercase.

2.1.4 Blocks

A block is a compound statement surrounded by braces ({}).[1] It can be used wherever the grammar of C requires a single statement. The common cases are the bodies of functions and **if**, **for**, **do**, **switch**, and **while** statements.

A block may also contain declarations. If it does, any declarations of **auto**, **register**, or **static** variables declare names that are local to the block. For example:

```
main()
{               /* OUTER BLOCK (BODY OF main FUNCTION) */
int i;
i = 1;
    .
    .
if (i == 1)
    {                   /* INNER BLOCK */
    float i;
    .
    .
    i = 3e10;
    }
}
```

In both blocks in the example, the variable i is declared with the default storage class **auto**. Within the inner block, i is a single-precision floating-point value; elsewhere, i is an integer. Since both declarations are of automatic variables, a new, floating-point version of i is allocated each time the inner block is activated.

If initialization is specified for any **auto** or **register** variables in a block, it is performed each time control reaches the block normally; that is,

---

1. Note that this use of the term block may differ from its use in other languages; in C, the terms block and compound statement are interchangeable.)

such initializations are not performed if a **goto** statement transfers control into the middle of the block or if the block is the body of a **switch** statement.

2.1.5 Comments

Comments, delimited by the character pairs /* and */, can be placed anywhere that white space can appear. The text of a comment can contain any characters except the close-comment delimiter (*/). Comments cannot be nested.

2.1.6 Keywords

Keywords are predefined identifiers. They cannot be redeclared. They identify C's data types, storage classes, and certain statements. Note that many conventional words in C programs are not actually keywords and can be redeclared. The notable examples are the names of functions, including main and the functions found in standard libraries that accompany C compilers.

Keywords must be written in lowercase letters.

Table 2–1 lists the C keywords.

Table 2–1: C Keywords

Keyword	Meaning
Type specifiers:	
int	Integer
long	Extended precision
unsigned	Unsigned integer
short	16-bit integer
char	8-bit integer
float	Single-precision floating-point number
double	Double-precision floating-point number
struct	Structure (aggregate of other types)
union	Union (aggregate of other types)
typedef	Tagged set of type specifiers
enum	Enumerated scalar type
void	None (reserved for future use)

Table 2–1: (Cont.) C Keywords

Keyword	Meaning
Storage-class specifiers:	
auto	Allocated at every block activation
static	Allocated at compile time
register	Allocated at every block activation
extern	Allocated by an external data definition (at compile time)
globaldef	Definition of global variable
globalref	Reference to global variable
globalvalue	Definition or declaration of global value
readonly	Allocated in read-only program section
Statements:	
goto	Transfers control unconditionally
return	Terminates a function and optionally returns a value to the caller
continue	Causes next iteration of containing loop
break	Terminates its corresponding **switch** or loop
if	Executes following statement conditionally
else	Provides an alternative for the **if** statement
for	Iterates the next statement (zero or more times) under control of three expressions
do	Iterates the next statement (one or more times) until a given condition is false
while	Iterates the next statement (zero or more times) while a given expression is true
switch	Executes one or more of the specified cases (multi-way branch)
case	Begins one case for **switch**
default	Provides default case for **switch**
entry	None (reserved for future use)
Operator:	
sizeof	Computes size of operand in bytes

Although they are not true keywords, the VAX–11 C compiler defines substitutions for the following identifiers; you should avoid redefining them:

```
vms
vax
vax11c
```

See Section 7.3 for more information on these identifiers.

2.2 External Data Definitions

An external data definition is one that occurs outside a function. It defines a name that can be used identically in several functions. The scope of the name is the remainder of the compilation. (That is, the scope is the remainder of the source file containing the definition, plus any files that are concatenated in the same compile command.) Thus, every function that follows the external data definition in the compilation can use the name as defined in the external definition, without declaring it.

External data definitions are syntactically the same as declarations of variables inside a function. A definition gives the data type of the variable(s), with a list of keywords (such as **short int**), the identifiers (which indicate whether the variable(s) are arrays or pointers), and any initial values. For example:

```
int x = 5; /* DEFINES A VARIABLE OUTSIDE A FUNCTION */
main()
{
/* THE FUNCTION CAN REFERENCE x WITHOUT DECLARING IT */
      printf("%d",x);
}
```

A function that precedes the external data definition can also reference the identifier by declaring it with the **extern** keyword. For example:

```
main()
{
/* DECLARES x AS EXTERNAL TO THE FUNCTION */
      extern int x;

      printf("%d",x);
}
/* THE DEFINITION OF x FOLLOWS THE REFERENCE TO IT */
int x = 5;
```

Furthermore, the external data definition can be in a separately compiled source file. The source file containing the definition and the source file containing the reference are linked together; the linker then resolves the external reference.

If no storage-class specifier appears in the external data definition, then the external definition creates a program section (psect) with the same name as the identifier and initializes it with the given initial value, if any. If a storage-class specifier does appear, it can be any storage class keyword except **auto** or **register**. Initializers can appear only when no storage class appears or when the storage class of the variable is **static**, **globalvalue**, **readonly**, or **globaldef**. (For more information on storage classes, see Chapter 3.)

Chapter 3

Data Types and Declarations

In C, data is represented by variables and constants. Every valid constant has a data type that is determined by the way in which the constant is written. Variables have data types specified in their declarations, and all variables must be declared. This chapter describes C's data types and the declaration of variables of these types. The general format of declarations is described in Section 3.1.

C recognizes the following fundamental, or scalar, data types. Each type describes the representation of a single datum. These types are described in detail in Section 3.2:

- Integers of various sizes. Integers are used in C to represent 8-, 16-, and 32-bit signed or unsigned numbers.
- Characters. C also has conventions and functions that facilitate the treatment of 8-bit integers as ASCII characters.
- Floating-point numbers, of either single or double precision.
- Pointers, which are variables containing 32-bit addresses that can point to functions and to variables of any other type, including other pointers and data structures.
- User-defined, or enumerated, types. Enumerated types are used to manipulate information in which a certain order is implicit, but for which a numeric representation is unnecessary or inappropriate.

Storage classes, identified in declarations by keywords such as **auto** and **static**, define the storage location and lifetime of variables. Storage classes are explained in Section 3.3.

Data structures, or aggregates, are made up of many members or elements, each of which is a fundamental type or an aggregate. The following data structures can be defined in C programs and are explained in detail in Section 3.4:

- Arrays, which are data structures made up of identically typed members, called elements.
- Structures, which are made up of members that may have different types and are stored in consecutive locations in memory.
- Unions, which are structures in which the members share the same storage.

Section 3.5 explains how to initialize variables in their declarations.

Section 3.6 explains the scope of names in C programs, that is, it explains whether a particular declaration permits a variable name to be used in several parts of a program or restricts its use to a particular part.

Section 3.7 suggests a model for interpreting complicated declarations and expressions written in C.

3.1 Format of a Declaration

A declaration is composed of the following items:

- A storage class keyword.
- At least one data type keyword, structure or union tag, **enum** tag, or **typedef** name, which gives the data type of the declared object.
- One or more declarators, which list the identifiers of the declared object and which may contain operators that declare a pointer, function, or array of objects of the declared type.
- At most one initializer for each declared object, giving the initial value of a scalar variable or the initial values of a structure, union, or array.

For example, the declaration

```
static int array[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

declares a static, 10-element array of integers, named array. The keyword **static** specifies the storage class; the keyword **int**, the data type. The declarator array[10] specifies an object with the identifier array, and the bracket operator ([10]) specifies an array size of 10 elements. The text "= { 1,...10 }" is an initializer that supplies the initial values of the 10 elements; the first element, array[0], is given the initial value 1, and so on.

3.2 Scalar Declarations and Types

In simple scalar declarations, each declarator is an identifier. For example:

```
int x,y,z;
```

declares three integers named x, y, and z. Since no storage class is specified, a default storage class is used. If this declaration appears outside any function, it is a *definition* of the external variables x, y, and z; that is, the variables are given the storage class **extern** by default. If the declaration appears within a function, x, y, and z are given the storage class **auto** by default.

The scalar data types — integers, characters, floating-point numbers, pointers, and enumerated types — are described in the subsections that follow.

3.2.1 Integers

Integers are declared with the keywords **int**, **short**, **long**, **char**, and **unsigned**, which specify the following internal representations:

- **int** and **long** specify a longword (32 bits) representing a signed integer. **int**, **long**, and **long int** can be used interchangeably in VAX-11 C. The range of possible **int** values is -2,147,483,648 to 2,147,483,647.
- **short** specifies a word (16 bits) representing a signed integer. **short** and **short int** can be used interchangeably. The range of possible **short** values is -32,768 to 32,767.
- **char** specifies a byte (8 bits) representing a signed integer. **char** variables can participate in arithmetic expressions with other integers. The numeric range of possible **char** values is -128 to 127. However, they usually represent ASCII characters.
- **unsigned** specifies a longword representing an unsigned integer. **unsigned** and **unsigned int** can be used interchangeably. The range of possible **unsigned** values is 0 to 4,294,967,295. **unsigned char** and **unsigned short** are also valid, meaning 8- and 16-bit unsigned integers, respectively. The range of possible **unsigned char** values is 0 to 255, and the range of possible **unsigned short** values is 0 to 65,535.

C also has integer constants in decimal, octal, and hexadecimal radixes. An integer constant is assumed to be decimal unless it begins with 0 or 0x; if it begins with 0, it is assumed to be octal; if it begins with 0x, it is assumed to be hexadecimal.

Integer constants must consist of the characters 0 to 9 and, optionally, the characters x, X, l, L, and -. (Because C has no unary plus operator, integer constants cannot include a plus sign.) In octal constants, the digits 8 and 9 have the octal values 10 and 11, respectively. Hexadecimal constants may also use either 'A' to 'F' or 'a' to 'f'.

Integer constants must not include a decimal point; constants with a decimal point are of type **double**.

Integer constants that exceed a longword are treated as programming errors and are flagged by the VAX-11 C compiler.

A decimal, octal, or hexadecimal integer immediately followed by an upper- or lowercase L is a long constant. Note, however, that **int** and **long** are identical data types in VAX-11 C, so the L suffix is not necessary.

Examples of *invalid* integer constants include:

```
143.                /* INCLUDES A DECIMAL POINT;
                       TYPE IS double */
-3333333333         /* OUT OF RANGE FOR int */
+33333              /* + IS AN INVALID CHARACTER */
77af                /* HEXADECIMAL CONSTANTS MUST BE
                       PREFIXED WITH 0x */
```

Note that **char** constants, such as 'a' and '$', are valid integer constants, too. Their integer values in VAX-11 C are the values of the corresponding ASCII codes.

3.2.2 Characters and Character Strings

A character variable is declared with the keyword **char**, and character constants are single characters enclosed in apostrophes, as in:

```
/* ch IS A CHARACTER VARIABLE */
char ch;

/* THE LOWERCASE LETTER 'a' IS
   A CONSTANT ASSIGNED TO ch */
ch = 'a';
```

A character-string variable is declared as an array of type **char**. A character-string constant is a series of characters enclosed in quotation marks and, in expressions, is treated as the address of the first character in the string. String constants cannot be directly assigned (with the assignment operator) to string variables, because arrays cannot be used on the left-hand side of the assignment operator. Instead, strings are copied with the **strcpy** and **strncpy** functions. (**strcpy** and **strncpy** are string-manipulating functions described in Chapter 6.) For example:

```
/* string IS A 10-ELEMENT ARRAY OF char */
char string[10];

/* COPY THE STRING constant INTO THE ARRAY string */
strcpy(string,"constant");
```

NOTE

The apostrophe (′) and quotation mark (") are significantly different punctuation marks in C, indicating a **char** constant and a string constant, respectively. One context in which the difference is important is in an argument list. If a function argument is specified as a string, then a constant argument for that function must be enclosed in quotation marks, not apostrophes, even if the string contains only one character.

A single character enclosed in apostrophes is a character constant, whose value is the ASCII code for the character. Nonprinting characters, the apostrophe, and the backslash are specified by the following escape sequences:[1]

Character	Mnemonic	Escape Sequence
newline	NL	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
apostrophe	′	\′
bit pattern	ddd	\ddd

Note that an escape sequence, such as '\n', denotes a single character.

The form \ddd is used to specify any byte value (usually an ASCII code), where ddd is one to three octal digits giving the character's value. (Here, the octal digits are limited to 0 to 7.) A common use is

```
'\0'
```

to specify the ASCII character NUL.

If the character following the backslash in an escape sequence is not one shown above, the backslash is ignored; that is, the character constant's value is the same as if the backslash were not present.

A string constant is a series of characters enclosed in quotation marks:

```
"This is a string."
```

It has the type "array of" **char** and storage class **static**. The string is initialized with the given characters and terminated (by the compiler) with a NUL ('\0') character. (The NUL is typically used by programs to find the end of a string.) Note that this representation means that

1. It is conventional in C programming to refer to these mnemonics as escape sequences. The term does not have the same meaning here as in "VT52 escape sequence" or other contexts in which it implies a string beginning with the ASCII character ESC.

there is no formal limit to the length of a string constant. The actual limit to a string constant's length in VAX-11 C is 1000 characters. All strings, even when written identically, are distinct objects.

When used in an expression, a string is treated as the address of the first character in the string. Thus, if p is a pointer to **char**, then

```
p = "thomasina";
```

is a valid expression that copies an address, not a string, to the pointer p. (Complete strings are transferred from place to place with the functions **strcpy** and **strncpy**.)

The following rules apply to the characters used in strings:

- All characters, including the escape sequences, can be used in strings.
- A quotation mark within a string must be preceded by a backslash (\).
- A backslash followed immediately by a newline is ignored, allowing long strings to be continued on the next line.

3.2.3 Floating-Point Numbers

The keyword **float** is used to declare a single-precision floating-point variable, represented internally in the VAX-11 F-floating binary format. The approximate range of absolute values for a **float** variable is 0.29×10^{-38} to 1.7×10^{38}. The approximate precision of a **float** variable is seven decimal digits.

The keyword **double** declares a double-precision floating-point variable, represented internally in VAX-11 D-floating binary format. (The keywords **double** and **long float** can be used interchangeably.) The approximate range of absolute values for **double** variables is the same as for **float**s, but the precision is approximately 16 decimal digits.

A floating constant has an integral part, a decimal point, a fractional part, an e or E, and an optionally signed integer exponent. The integral and fractional parts consist of decimal digits; either the integral or fractional part may be omitted. Either the decimal point or the "E<exponent>" (but not both) may be omitted.

All floating constants are of type **double**.

Some examples of floating constants are:

```
3.0e10
3.0E-10
3.0e+10
-3E10
3.0
.120e2
.120
```

Floating-point constants in C can be unsigned or negative; note again that, because C has no unary plus operator, a constant such as

```
+3.0e10
```

is *not* valid.

3.2.4 Pointers

Pointers in C are variables that contain addresses of other variables. They are not declared with a keyword, but instead are declared with an asterisk, as in:

```
int *px;
```

The identifier px is declared as a pointer to a variable of type **int**.

Pointers always contain the addresses of known objects or are null pointers. A null pointer is a pointer variable that has been assigned the integer constant 0, and does not point to any object.

A declarator of the form

```
*declarator
```

is used to declare a pointer.

The declaration

```
int *intptr, i;
```

declares intptr as a pointer to an integer and i as an integer.

Pointers are declared as pointing to a particular data type. The "type of the pointer" (here, **int**) is used when the pointer participates in certain arithmetic operations with nonpointer expressions. Furthermore, some contexts, such as argument lists, may require a pointer of a particular type.

The unary asterisk (*) is also the indirection operator in C. For example:

```
i = *intptr;
```

assigns an integer (the value of the object pointed to by intptr) to i. Since the asterisk can be used in any sort of declarator, you can have pointers to scalars, to functions, to other pointers, to structures, and so forth.

The name intptr can also be used without the asterisk operator to represent the pointer variable, rather than the object it points to. For example:

```
intptr = &x;
```

assigns the address (using the "address-of" unary operator &) of the variable x to intptr.

3.2.5 Enumerated Types

An enumerated type is a data type that is not derived from other fundamental types. For example, a type named spectrum can be enumerated by writing:

```
enum spectrum
    {
    red,orange,yellow,green,blue,indigo,violet
    };
```

where spectrum is the enumeration tag of the new type, and red, orange, ...violet are the enumerators. These enumerators are the constant values of the type spectrum and can be used wherever constants are valid. This declaration merely associates the tag spectrum with the list of constants and does not declare any data of type spectrum.

Data of type spectrum could be declared by writing a list of identifiers after the type enumeration

```
enum spectrum
    {
    red,orange,yellow,green,blue,indigo,violet
    } color1;
```

or by using the tag spectrum as a reference to the type enumerated elsewhere, as in:

```
enum spectrum color1;
```

Both examples declare color1 as an object of type spectrum — that is, a spectrum variable. The second form must occur within the scope of the definition of spectrum.

An enumerated type also can be declared with no tag, as in:

```
enum
    {
    out,verydim,dim,prettybright,bright
    } light;
```

This declaration defines a variable (light) of an enumerated type. The variable can assume any of the enumerated values, but since there is no enumeration tag like spectrum, other declarations cannot refer to this type without replicating the entire list of constant values: out, verydim, ... bright.

enum tags (such as spectrum) can have the same spellings as other identifiers in the same program (including variables and member names in structures and unions), because the meanings are distinguished by context. Tags are subject to the same scope rules as other identifiers. However, **enum** constant names must be spelled uniquely.

In declarations of enumerated types, the place of a data type keyword is taken by an enum-specifier of the form:

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

where the identifier is the name, or tag, of the new type. The enum-list gives an ordered list of values that a datum of this type can have. Thus:

enum-list ::=
    enumerator
    enum-list , enumerator

where

enumerator ::=
    identifier
    identifier = constant-expression

Internally, each enumerator is associated with an integer constant; the first enumerator in the enum-list is given the value zero by default, and the enumerators are incremented by one as they are read from left to right. Any enumerator can be followed by the syntax "= constant-expression" to set it to a specific (integer) constant value. The enumerators to the right of such a construct (unless they have constant expressions, too) then receive values that begin at constant-expression+1. For instance:

```
enum spectrum
    {
    red,yellow=2,green,blue,indigo,violet
    };
```

gives red, yellow, green, ... the values 0,2,3, ...

Enumerated data types should be regarded as distinct from the fundamental types, although they are stored internally as integers. Examining the value of an object like spectrum above (by writing it out, for instance) shows an integer, not a string such as "red" or "yellow."

Type mismatches between the enumerated and fundamental types, or between different enumerated types, are considered errors. Thus, it is not valid to say

```
enum spectrum
     {
     red,orange,yellow,green,blue,indigo,violet
     } color1;
enum illum
     {
     out,verydim,dim,prettybright,bright
     } light;
     .
     .
light = red;
```

because red and light have different enumerated types. Nor is it valid to say

```
enum
     {
     out,verydim,dim,prettybright,bright
     } light;
     .
     .
light = 1;
```

because 1 is not an enumerated value for light.

To make mixed-type operations valid, use the cast operator. For example:

```
light = out + (enum illum) red;
     .
     .
light = (enum illum) 1;
```

Here, the cast (enum illum) causes the compiler to treat the **enum** constant red and the integer constant 1 as values of the enumerated type illum. (For general information on the cast operator, see Section 4.3.5.)

3.3 Storage Classes

The storage class of a name determines its location and scope. The C storage classes are:

- **auto**, indicating that storage is allocated at the activation of the defining block (that is, at run time) and exists only for the duration of that block activation.

- **static**, indicating that the storage is allocated at compile time and exists for the duration of the program. **static** variables reside in the program section (psect) named $DATA; however, **static readonly** variables reside in the program section named $CODE.
- **register**, indicating (to C compilers in general) that the variable is to be placed in a machine register, if possible (only function parameters and automatic variables can have this class).

  In VAX-11 C, registers are allocated based on frequency of use of a given variable. The keyword **register** is ignored by the VAX-11 C compiler. Any scalar variable with the storage class **auto** or **register** is eligible for allocation to registers as long as its address is not taken with the ampersand operator and it is not a member of a structure or union.
- **extern**, indicating a reference to storage defined elsewhere in an external data definition. No storage is allocated by an **extern** declaration, and thus no initializers can appear in it.

  In VAX-11 C, each **extern** variable is assigned a separate program section with the same name as the variable. If **readonly** appears, the program section has the NOWRT (not writeable) attribute. There can be approximately 65,532 **extern** names per compilation.
- **globaldef**, defining the variable as a global symbol.
- **globalref**, indicating that the variable is a global symbol that is defined elsewhere.
- **readonly**, used with **static**, **extern**, and **globaldef**. It assigns the variable in a program section with the attribute NOWRT. The keyword **readonly**, used by itself, implies **extern**.
- **globalvalue**, indicating that the declared object is a global name for an integer (or **enum** value).

For more details on **globalref**, **globaldef**, **globalvalue**, and **readonly**, see Chapter 10, Storage Allocation, and Chapter 11, Global Symbols.

The **auto** and **register** classes are local, or internal, to their defining blocks. **static** names may be known internally or externally, depending on whether the declaration of the object is inside or outside a function definition. **extern** names are known everywhere within the scope of their external definitions.

As with many languages, VAX-11 C programs are often constructed of separately compiled modules. Each module can contain several function definitions and several external data definitions. It is primarily in this case that the difference between **static** and **extern** is significant. In an external data definition, a name can be declared with either **static** or **extern**; however, a **static** name is known only in the remainder of its own module, whereas an **extern** name is known everywhere in the program. For more information, see Section 10.1.

If no storage-class specifier appears in a declaration, the defaults are as follows:

- Inside a function, the default is **auto** (except for function identifiers).
- Outside a function (that is, in an external data definition), the construct is considered to be the definition of an **extern** variable. **globaldef** and **globalvalue**, which are VAX-11 C extensions, are never taken as defaults.

extern, **globaldef**, and **static** variables are initialized to zero by default (and only once) if no initialization is specified explicitly.

An explicitly initialized **auto** or **register** variable is initialized each time its declaring function or block is activated normally (that is, control is transferred into the block by some means other than a **goto** statement). There is no default initialization of such variables.

3.4 Data Structures

The aggregate data types — arrays, structures, and unions — are described in the sections that follow.

3.4.1 Arrays

Arrays are declared with square brackets ([]). The elements of an array can be other arrays (to form multidimensional arrays), structures, unions, scalars of all types, and pointers to any object.

If an array name appears in the argument list of a function, the address of the beginning of the array is passed. Therefore, subscripted references in the called function can modify elements of the array.

A declarator of the form

declarator[*constant-expression*]

is used to declare an array. The constant expression gives the number of elements in a single dimension. Array subscripts in C begin with 0, not 1, and they must be integral. Therefore, the constant expression gives the number of elements in a dimension, but not the maximum subscript. That is,

```
int arrayint[10];
```

declares an array of 10 integers; the maximum subscript is 9.

A multidimensional array is declared as an array of arrays, for example:

```
int array2[10][2];
```

where array2 is a two-dimensional array containing 20 integers. The same syntax is used to reference an individual element, as in:

```
++array2[0][0]; /* INCREMENT FIRST ELEMENT. */
```

An element of a multidimensional array should not be referenced with multiple expressions in the same set of brackets; in fact, since the comma is an operator, the reference to array[i,j] is the same as a reference to array[j].

In C, arrays are stored in row-major order (the rightmost subscript varies most rapidly).

The constant expression is optional only in the first pair of brackets after the array's identifier. Omission of the constant expression is useful in the following cases:

- If the array is external, its storage is allocated by a remote definition. Therefore, the constant expression can be omitted for convenience when the array name is declared, as in:

```
extern int array1[];
function_1()
{
.
.
.
}
```

  In a separate compilation:

```
int array1[10];
function_2()
{
.
.
.
}
```

- If the declaration of the array includes initializers, the size of the array can be omitted; it is calculated from the number of initializers.
- If the array is used as a function parameter, it is defined in the calling function. The declaration of the parameter in the called function can omit the constant expression. Example 3–1 shows how an array is used in this way.

```
main()
{
/* STRING TO BE PASSED AS ARGUMENT IS DECLARED AS
   A POINTER TO CHAR */
    char *s = "Thomas";
    int sum;
    sum = adder(s);
}
/* ADD UP ASCII VALUES */
adder(string)
/* DECLARATION OF PARAMETER OMITS THE CONSTANT
   EXPRESSION */
char string[];
{
/* SUM IS INITIALLY ZERO */
    int i,sum=0;
    for (i=0; string[i] != '\0'; i++) sum+=string[i];
    return sum;
}
```

Example 3-1: Using Arrays as Function Parameters

When the function adder is called, the parameter "string" receives the address of the first character of the argument s, which can then be manipulated in adder. The declaration

```
char string[];
```

serves only to give the type of the parameter, not to reserve storage for it.

3.4.2 Structures and Unions

Structures and unions are declared by the keywords **struct** and **union**, respectively. The members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0.

In a union, every member begins at offset 0 from the address of the union; that is, each member of a union denotes the same storage. Unions cannot be initialized.

Structures and unions share the following characteristics:

- Their members can be variables of any type, including other structures and unions or arrays. A member can also consist of a specified number of bits, called a field.
- They can be assigned to other structures and unions with the simple assignment operator (=). The two structures or unions in the assignment must have the same length.

- They can be passed as arguments that correspond to structure or union parameters, and returned by functions. The two structures or unions involved in argument passing must have the same length. A structure or union is passed by value, just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.
- The only operators that are valid with structures and unions are simple assignment (=) and **sizeof**. In particular, structures and unions may not appear as operands of the equality (==), inequality (!=), or cast operator.

Structures and unions are declared with the **struct** or **union** type specifier:

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

Except for the keywords **struct** and **union**, these aggregates are declared with identical syntax. The optional identifier in the above syntax is known as the tag of the structure or union. The forms of declaration are used as follows:

1. If the tag is omitted, the structure or union definition applies only to the identifiers that follow it in the declaration.
2. If a declaration includes both the tag and struct-decl-list, then it declares one or more identifiers to be variables with the given structure, and it declares the tag to be a shorthand, or mnemonic, notation for the structure.
3. The third form above uses the tag to refer to a previously defined structure or union. The definition is then applied to the identifiers that follow the struct-or-union-specifier in the declaration.

The following examples illustrate the three forms of structure declaration:

```
/* FIRST FORM; STRUCTURE WITH NO TAG */
struct
    {
    char first[20]; /* FIRST NAME. */
    char middle[2]; /* MIDDLE INITIAL AND PERIOD. */
    char last[30];  /* LAST NAME. */
    int iq;         /* IQ. */
    } tom,mary ;    /* AND TWO VARIABLES WITH
                       THIS STRUCTURE. */
```

Each successive nonfield member of a structure begins at the next byte boundary; there is no implicit type alignment. For example, the member iq is not necessarily aligned on a longword boundary, even though it is an **int.**[1]

A reference to a member of a structure must be fully qualified, or it must be a pointer-qualified reference (discussed later). For example, the reference to Tom's IQ is:

```
tom.iq
```

A member name denotes the member's data type and its offset from the base of the structure. There are no restrictions on the reuse (as a member name) or redeclaration of a particular name except that the same name cannot be used for more than one member in the same structure. A member name is unique if it conforms to either of the following requirements:

- It is used only once.
- If it is used more than once (in different structures), every use denotes a member of the same data type and at the same offset from the base of its structure.

Member names are normally used to refer to the same structure or union in which the member name was declared. The following checks apply to the use of member names for reference to structures and unions in which they were not declared:

- If a member name is unique, you can use it in a reference to a structure of which it is not a member, since the address and size of the referenced datum can be determined without ambiguity. However, the compiler issues a nonfatal warning message. (This usage is maintained for compatibility with old C programs.)
- If a member name is not unique (ambiguous), its use in such a reference causes a fatal error message.

In VAX-11 C, and in other recent compilers, a structure or union reference must be uniquely qualified; that is, a member name in a reference must be prefixed either with a pointer qualifier (pointer-name ->) or with the name of the structure or union and the names of all intervening members that are required to make the reference unambiguous. For example, consider the following structure declaration:

1. This alignment of structure members is a VAX-11 C convention and is followed by all other VAX-11 languages. Other C implementations may align members differently.

```
main()
{
     struct
          {
          struct { int a1,a2,a3; } mema;
          struct { int a1,a2,a3; } memb;
          } *pointer,structure;

     pointer = &structure;

     structure.mema.a1 = 1;   /* NONAMBIGUOUS */
     pointer->memb.a1 = 2;

     structure.a1 = 3;        /* AMBIGUOUS */
     pointer->a1 = 4;
}
```

A reference to one of the integers in this structure must be of the form:

```
structure.mema.a1
pointer->mema.a1
```

but not:

```
structure.a1
pointer->a1
```

In fact, structure members that are themselves structures must be given member names (as with mema above) to make it possible to construct fully qualified references.

If the structure has a tag, then the tag can be used to declare more variables with the same structure. For example:

```
/* SECOND FORM; STRUCTURE WITH THE TAG person */
struct person
     {
     char first[20]    /* FIRST NAME */
     char middle[2];   /* MIDDLE INITIAL */
     char last[30];    /* LAST NAME */
     int iq;           /* IQ */
     };  /* NO VARIABLES DECLARED HERE,
            JUST THE STRUCTURE AND ITS TAG */

/* THIRD FORM: TWO STRUCTURE VARIABLES
               OF TYPE person */
struct person dick,Jane;
```

Structure tags can have the same spellings as other variables. The compiler distinguishes them by context. Structure tags can also have the same spellings as member names.

Structures can contain other structures. For example:

```
struct date
     {
     int day;
     int month;
     int year;
     int yearday;
     char month_name[4];
     };

struct person  /* STRUCTURE WITH THE TAG person */
     {
     char first[20];          /* FIRST NAME */
     char middle[2];          /* MIDDLE INITIAL */
     char last[30];           /* LAST NAME */
     int iq;                  /* IQ */
     struct date birth;       /* DATE OF BIRTH */
     struct date election;    /* DATE OF ELECTION */
     };  /* NO VARIABLES ARE DECLARED */

/* DECLARE dick AND jane AS STRUCTURES OF TYPE
   person WITH date OF birth AND election */
struct person dick,jane;
```

In a structure or union declaration, the keyword **struct** or **union** (and the tag, if there is one) is followed by a a pair of braces ({}) that enclose a list of the form:

```
struct-decl-list ::=
        struct-declaration
        struct-declaration struct-decl-list
```

Each struct-declaration describes an individual member, giving its type and other attributes:

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

where struct-declarators have the form:

```
struct-declarator ::=
    declarator
    declarator : constant-expression
    : constant-expression
```

Usually, a struct-declarator is just a declarator for the member of the structure or union, as in the previous examples where the declarator

```
char first[20];
```

specified a member that was a 20-element array of **char**.

In structure declarations, initializers follow the structure variables, not the members. Consider, for example:

```
struct
    {
    int i;
    float c;
    } a = { 1,3.0e10 }, b = { 2,1.5e5 };
```

This example declares the structure variables a and b with different initial values.

A structure member may also consist of a specified number of bits, called a field, which may be either named or unnamed. A colon is used to separate the member's declarator (if any) from a constant-expression that gives the field width in bits. No field may be longer than 32 bits in VAX-11 C.

If no field name precedes the field-width expression, the struct-declarator indicates an unnamed field of the specified width. Note that since nonfield structure members are aligned on byte boundaries, this form can create unnamed gaps in the structure's storage. As a special case, an unnamed field of width zero causes the next member (generally another field) to be aligned on a byte boundary.

The use of field types other than **unsigned** or **int** is an error. There are no restrictions on the use of fields except as follows:

- There can be no arrays of fields.
- The "address of" (&) operator cannot be applied to fields, and consequently there cannot be pointers to fields.

Scalar items (except fields), arrays, structures, unions, and **enum** members are aligned on the next byte boundary. Sequences of fields are packed as tightly as possible. On the VAX-11, fields are assigned from right to left.

For example, the following structure declaration results in the alignments shown in Figure 3-1:

```
static struct
    {
    char c;
    short int i;
    unsigned fld1 : 3;
    unsigned fld2 : 4;
    unsigned      : 0;
    unsigned fld3 : 4;
    } a = { 'A', 1024, 06, 012, 014 } ;
```

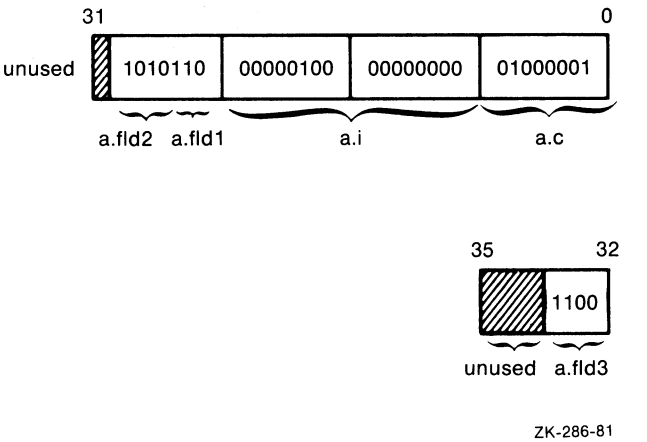


Figure 3-1: Alignment of Structure Members

In Figure 3-1, the member a.i is aligned on the second byte (at bit 8), because scalar, nonfield members are aligned on byte boundaries. Notice that the fields a.fld1 and a.fld2 are packed as tightly as possible in the high-order byte of the first longword. The unnamed, zero-length field preceding a.fld3 causes that field to be aligned on the next byte boundary, bit 32.

You can use the /MAP qualifier on the LINK command to produce a storage map. The storage map shows how structure members have been aligned by the linker.

3.5 Initialization

In declarations, variables can be given initial values with an initializer. An initializer consists of an equal sign (=) followed by either a single expression or a comma-separated list of one or more expressions in braces.

For **static** and **extern** variables, all expressions in an initializer must be constant expressions or must give the address of a previously declared variable, possibly offset by a constant expression.

For **auto** and **register** variables, the expressions in an initializer may be arbitrary expressions involving previously declared variables, constants, and functions. To be used in initializers, variables and functions must have known values. That is, a variable must have been initialized or assigned a value, and a function must return a value.

3.5.1 Initialization of Scalar Variables

An initializer that applies to a scalar variable contains a single expression, possibly enclosed in braces. The evaluated expression is used as the scalar's initial value, with the same conversions performed as for an assignment to the type of the scalar. For example, the declarations

```
int i = 1;
char ch = 'a';
float c = 3.0e10;
float f = 1;
```

initialize the variables i, ch, c, and f with 1, 'a', 3.0e10, and 1, respectively. At compile time, the integer constant 1 is converted to **float** before it is used to initialize f.

3.5.2 Initialization of Aggregate Variables

Initializers are assigned to an array or structure in row-major order or increasing member order. If there are fewer initializers than members for a **static** aggregate, the aggregate is padded with zeros. The following rules govern the use of braces in initializer lists for structures and arrays:

- If the initializer for an aggregate begins with a left brace ({), then the following comma-separated list provides initial values for the aggregate's elements or members. The list of initializers can end with a comma, which is ignored. The number of initializers cannot be greater than the number of elements or members.
- If the initializer does not begin with a left brace, then only enough elements are taken from the initializer list to supply values to the aggregate's members. In this case, there can be more initializers than there are elements or members, and any remaining values in the list are left to initialize the next aggregate.

For example, the declaration:

```
static int x[4][3] =
    {
    { 1,2,3 },
    { 4,5,6 },
    { 7,8,9 },
    };
```

initializes the first nine elements of the 12-element (two-dimensional) array x: x[0][0] is initially 1, x[0][1] is 2,...x[2][2] is 9. That is, each inner set of braces matches one row of the array. So, the initializer for row x[0] begins with a left brace, and the three initial values following the brace initialize that row's elements. There cannot be more than three initial values. No initializers are given for the last row, row x[3], so, because x

has storage class **static**, x[3][0],...x[3][2] are initialized with zero. The comma following the last initializer ({ 7, 8, 9 }) is ignored. This declaration has the identical effect as:

```
static int x[4][3] = { 1,2,3,4,5,6,7,8,9 };
```

In this case, although the initializer for x begins with a left brace, the initializer for row x[0] does not; therefore, the first three initial values are taken for x[0][0],...x[0][2], and the leftover values remain for the next aggregate. The next aggregate is row x[1]; again, that row's initializer does not begin with a left brace, so the values 4, 5, and 6 initialize the elements in that row.

If the initializers for the rows do not specify enough values for each column, then the elements corresponding to the missing values are initialized with zero. For example:

```
static int x[4][3] =
    {
    { 1,2 },
    { 4,5 },
    { 7,8 },
    };
```

initializes x[0][0] with 1, x[0][1] with 2, x[1][0] with 4,..., x[2][2] with 8. The elements in column 2, such as x[0][2], are initialized with zero, as are all the elements in the row x[3].

When the size of an array is omitted from a declaration, the compiler determines the size by counting the initializers. For example:

```
int bins[] = {1,2,3,4,5};
```

Here, the array bins is given a size of 5 elements.

An array of characters can be initialized with a string constant as long as the storage class of the array is not **auto**. In that case, the assumed size includes the NUL character which terminates all strings. For example:

```
static char name[] = "Wilbur";
```

makes "name" an array of 7 characters, and is equivalent to

```
char name[] = {'W','i','l','b','u','r','\0'}
```

or

```
char name[7] = {'W','i','l','b','u','r','\0'}
```

There is no way to specify iterations of an initializer or to initialize an element or member in the middle of an aggregate without also initializing the previous elements or members.

Example 3-2 shows these rules applied to an array of structures.

```
main()
{
     int l,m;
     static struct
          {
          char ch;
          int i;
          float c;
          } ar[2][3] =
             {
     /* INITIALIZER FOR ROW ar[0]: */
             {{'a', 1, 3e10},
             {'b', 2, 4e10},
             {'c', 3, 5e10}},
     /* NO INITIALIZER FOR ROW ar[1] */
             };
     for (l = 0; l < 2; l++)
          for (m = 0; m < 3; m++)
             {
             printf("first member, row %d col %d: %c\n",
              l,m,ar[l][m].ch);
             printf("second member, row %d col %d: %d\n",
              l,m,ar[l][m].i);
             printf("third member, row %d col %d: %e\n",
              l,m,ar[l][m].c);
             }
}
```

This program writes the following output to **stdout**:

```
first member, row 0 col 0: a
second member, row 0 col 0: 1
third member, row 0 col 0: 3.000000e+10
first member, row 0 col 1: b
second member, row 0 col 1: 2
third member, row 0 col 1: 4.000000e+10
first member, row 0 col 2: c
second member, row 0 col 2: 3
third member, row 0 col 2: 5.000000e+10
first member, row 1 col 0:
second member, row 1 col 0: 0
third member, row 1 col 0: 0.000000e+00
first member, row 1 col 1:
second member, row 1 col 1: 0
third member, row 1 col 1: 0.000000e+00
first member, row 1 col 2:
second member, row 1 col 2: 0
third member, row 1 col 2: 0.000000e+00
```

Example 3-2: Arrays of Structures

3.6 Scope of Names

The scope of a name is that portion of a program in which the name has meaning. The following scope rules apply to the names of **enum** constants, **typedef** names, variables, and functions:

- The scope of a name defined in an external data definition is the remainder of the current compilation. That is, the scope is the remainder of the source file plus any source files that are concatenated in the compiler command line after the source file containing the definition. The scope can be overridden by a declaration in a subsequent function or block.
- The scope of a name defined in a block is limited to that block. The definition can be overridden by a declaration in an internal block.
- The scope of a parameter name is the entire function in which the name is declared. The definition can be overridden by a declaration in an internal block.
- Function names are implicitly declared as **extern** when an undeclared function is called.

There are other categories of names that can be used without conflict with variable and function names or with each other. These are:

- The names of labels. Labels are used only as the targets of **goto** statements, and that context distinguishes them from variables of the same name. The scope of a label name is the entire function in which the label appears.
- The tags used in **struct** and **union** declarations. Two structures or **enum** types cannot have the same tag, but the tags can be the same as the identifiers used for variable and function names. The scope of tags is the same as the scope of the declarations in which they appear. (Note that **enum** constant names, unlike **enum** tags, must be distinct from the names of variables or functions in the same scope.)
- The names of structure or union members. The scope of member names is the same as the scope of the declarations in which they appear.

3.7 Interpreting Declarations

The C programming language syntax for declaring objects is rather unlike the declaration syntax of other languages. Since the exact meaning of a complicated C declaration is not always immediately apparent, even to an experienced C programmer, this section gives guidelines for interpreting (or, possibly, constructing) C declarations.

C uses the same set of operators and symbols for declarators as for identifiers in an expression. For example:

```
int x;
int *px;
```

declare an integer, x, and a pointer to an integer, px. The declarator *px has the same form as that used to yield an integer in an expression, such as:

```
x = *px;
```

In the case of simple declarators, this symmetry makes it fairly easy to determine the type of an expression or the meaning of a declarator.

More complicated declarators can be more difficult to interpret without some additional guidelines. The important one to remember is that the symbols used in declarators are C operators, subject to the usual rules of precedence and grouping (associativity). In order of precedence, the operators used in declarators are:

1. The primary-expression operators ( ) for "function returning..." and [] for "array of...", where the ellipsis indicates the type specified in the declaration. These operators group from left to right.
2. The unary asterisk (*), for indirection or "pointer to...", which groups from right to left.

Consider, for example:

```
int *x[];
```

Even this brief declaration may be confusing. Does it declare an array of pointers to integers, or a pointer to an array of integers? Since the brackets are of higher precedence, it follows that:

1. *x[] is an integer
2. x[] is a pointer to an integer
3. x is an array of pointers to integers

Most complicated declarators and expressions can be interpreted fairly quickly by such a sequential breakdown. Note that the asterisk was removed before the brackets because it is of lower precedence.

Also note that this interpretation process has the desirable property that it enumerates all the possible usage constructs involving a declarator and gives the semantic interpretation.

When constructing or interpreting declarations or expressions, use the following scheme[1] for translating operators to English and vice versa:

- "*" == "pointer to"
- "( )" == "function returning"
- "[ ]" == "array of"

1. Bruce Anderson, "Type Syntax in the Language C: An Object Lesson in Syntactic Innovation," *SIGPLAN Notices* 15, No. 2 (March 1980).

For a more interesting example, consider:

```
char *x()[];
```

The breakdown is:

1. `*x()[]` is **char**
2. `x()[]` is (pointer to) **char**
3. `x()` is (array of) (pointer to) **char**
4. `x` is (function returning) (array of) (pointer to) **char**

In step 3, the `[]` operator is removed first because primary-expression operators are of equal precedence and group from left to right. That is to say, "`()[]`" means "function returning array of", not "array of function returning...".

As a general rule, when breaking down a declaration this way, remove the operators with the lowest precedence first. Then, if operators are of equal precedence and group from left to right, remove the rightmost operator first; if they group from right to left, remove the leftmost operator first.

As it happens, the declaration shown above is semantically invalid; C allows functions returning addresses of arrays, but not functions returning arrays. Perhaps the intention was a function returning the address of an array of pointers to characters. The declaration can be made valid by starting at the bottom of a breakdown and working back up to a valid declaration:

1. `x` is (function returning) (pointer to) (array of) (pointer to) **char**
2. `x()` is (pointer to) (array of) (pointer to) **char**
3. `*x()` is (array of) (pointer to) **char**
4. `(*x())[]` is (pointer to) **char**
5. `*(*x())[]` is **char**
6. `char *(*x())[];`

In the final declaration, the first asterisk (since it groups right to left) applies to **char**. Clearly, such a declaration, once it is known to have the desired meaning, should have a comment explaining its purpose.

Parentheses (in addition to the ( ) operator) are used in declarations to change the binding of operators. For example, the outer parentheses introduced in step 4 of the previous example prevent the brackets from binding to the inner set of parentheses.

As a last case, consider:

```
char (* (*x()) []) ();
```

This means:

1. `(* (*x()) []) ()` is **char**
2. `* (*x()) []` is (function returning) **char**
3. `(*x()) []` is (pointer to) (function returning) **char**
4. `*x()` is (array of) (pointer to) (function returning) **char**
5. `x()` is (pointer to) (array of) (pointer to) (function returning) **char**
6. `x` is a function returning a pointer to an array of pointers to functions returning characters

Spaces were used in the example to separate the declarator into its component parts. Since spaces, tabs, and newlines are ignored by the parser, they can and should be used in actual declarations for clarity.

3.8 typedef

The keyword **typedef** is used to define an abbreviated name, or synonym, for a lengthy type definition. Grammatically, the word **typedef** is a storage-class specifier, so it can precede any valid declaration. In such a declaration, the identifiers name types instead of variables. For example:

```
typedef char CH, *CP, STRING[10],CF();
```

In the scope of this declaration, CH is a synonym for "character," CP for "pointer to character," STRING for "10-element array of characters," and CF for "function returning a character." Each of the type definitions can be used in that scope to declare variables, as in:

```
CF c;      /*c IS A FUNCTION RETURNING A CHARACTER*/
STRING s; /*s IS A 10-CHARACTER STRING*/
```

Chapter 4

Expressions and Operators

An expression is any series of symbols that C uses to produce a value. The simplest expressions are constants and variable names. They have no operators and they yield a value directly. Other expressions combine operators and subexpressions to produce values.

This chapter describes the following aspects of expressions and operators:

- Data type conversions
- Primary expressions and operators
- Unary expressions and operators
- Binary expressions and operators
- Assignment expressions and operators
- The conditional expression and operator
- The comma expression and operator

Table 4–1 shows the set of C operators arranged by precedence. Those operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Operators of equal precedence appear in the same row.

For example, in the expression

```
A*B+C
```

A and B are multiplied first, because * is of higher precedence than +. The table also includes the associativity rule that applies to each row of operators. That is, the expression

```
A/B/C
```

is evaluated as

```
(A/B)/C
```

because the / operator groups from left to right.

As Table 4–1 shows, the operators fall into the following categories:

- Primary operators, which usually modify or qualify identifiers. For example, both the arrow operator (->) and the period operator qualify structure references.
- Unary operators, which take a single operand. A familiar example is the unary minus sign, which negates its arithmetic operand.

Table 4-1: Precedence of C Operators

Category	Operator	Associativity
primary	( ) [ ] -> .	left to right
unary	! ~ ++ -- - (type) * & sizeof	right to left
binary (mult.)	* / %	left to right
binary (add.)	+ -	left to right
binary (shift)	<< >>	left to right
binary (relat.)	< <= > >=	left to right
binary (equal.)	== !=	left to right
binary (bitand)	&	left to right
binary (bitxor)	^	left to right
binary (bitor)	|	left to right
binary (AND)	&&	left to right
binary (OR)	||	left to right
conditional	?:	right to left
assignment	= += -= *= /= %= >>= <<= &= ^= |=	right to left
comma	,	left to right

- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which, in the expression

  ```
  A ? B : C;
  ```

  evaluates either expression B or C, based on the evaluation of expression A.
- Assignment operators, which assign a value to a variable, optionally performing an additional operation before the assignment takes place.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

To understand the details about particular operators, you must also understand the circumstances in which C performs data type conversions.

4.1 Data Type Conversions

C performs data type conversions in three situations:

1. When two or more operands of different types appear in an expression (including an assignment).
2. When arguments other than long integers, addresses, or double-precision floating-point numbers are passed to a function.
3. When the data type of an operand is deliberately converted by the cast operator. (The cast operator is described in Section 4.3.5.)

4.1.1 Conversion of Operands

The following rules (sometimes referred to as the usual arithmetic conversion rules) govern the conversion of operands in arithmetic expressions. Although they do not specify explicit conversions at the machine-language level, the rules govern in the following order:

1. Any operands of type **char** or **short** (signed or unsigned) are converted to their 32-bit equivalents (**int** or **unsigned int**), and any of type **float** are converted to **double**.
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**, the other is converted to **unsigned**, and that is the type of the result.
4. Otherwise, both operands must be **int**, and that is the type of the result.

The usual arithmetic conversions are performed on all arithmetic operands. Note that some operators (such as the shift operators >> and <<) require integers as operands, and this requirement cannot be met if one operand is of type **float** or **double**.

In general, floating-point arithmetic is carried out in double precision. Whenever an operand of type **float** appears in an expression, it is converted to type **double**; the compiler lengthens the operand by padding its fractional part with zeros.

When an operand of type **double** is converted to **float** — for example, by an assignment — the operand is rounded before being truncated to **float**.

A **float** or **double** value operand may also be converted to an integer by assignment to an integral variable. In VAX-11 C, the truncation of the **float** or **double** value is always toward zero.

Conversions also take place between the various kinds of integers. In VAX-11 C, **char**s are bytes treated as signed integers. When a longer

integer is converted to a shorter integer or to **char**, it is truncated on the left; excess bits are discarded. For example:

```
int i;
char c;

i = 0xFFFFFF41;
c = i;
```

assigns hex 41 ('A') to c. Shorter signed integers are converted to longer ones by sign extension.

Whenever an unsigned integer and a signed integer are combined, the signed integer is converted to **unsigned** and the result is **unsigned**. All conversions from signed to unsigned perform an intermediate conversion to **int**. For example, the compiler converts a **char** or **short** operand to an unsigned version by first converting it to a signed **int** and then by truncating it to form the unsigned version. All conversions from unsigned to signed (such as by the cast operator) involve an intermediate conversion to **unsigned int**.

You can also add integers to pointers, in which case the integer is scaled (multiplied) by a factor that depends on the type of the object to which the pointer points. For more details, see the discussion of the additive operators (Section 4.4.1).

4.1.2 Conversion of Function Arguments

The data types of function arguments are assumed to match the types of the formal parameters. C does not compare the types case by case. Instead, all arguments of type **float** are converted to **double**, all **chars** and **shorts** are converted to **ints**, all **unsigned chars** and **unsigned shorts** are converted to **unsigned ints**, and an array or function name is converted to the address of the named array or function. No other conversions are performed automatically, and any mismatches after these conversions are programming errors. Use the cast operator to pass arguments to parameters of different types.

4.2 Primary Expressions and Operators

Primary expressions denote values. Primary expressions include previously declared identifiers, constants (including strings), array references, function calls, and structure or union references. Syntactically, the primary expressions are as follows:

```
primary ::=
        identifier
        constant
        string
        ( expression )
        primary ( expression-list )
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
```

The simplest forms are identifiers (variable names) and string or arithmetic constants. Other forms are parenthesized expressions, function calls, array references, lvalues and rvalues (see Section 4.2.4), and structure and union references.

4.2.1 Parenthesized Expressions

An expression within parentheses has the same type and value as the same expression without parentheses. As in declarations, any expression can be parenthesized to change the grouping, or associativity, of its operators.

4.2.2 Function Calls

A function call is a primary expression followed by parentheses. The parentheses may contain a list of arguments (separated by commas) or may be empty. An undeclared function is assumed to be a function returning **int**. If an identifier was declared as a "function returning...", but is used in a context other than a function call, it is converted to "address of function returning...". That is, the declaration

```
double atof();
```

declares a function returning **double**. The name **atof** can then be used in a function call:

```
result = atof(c);
```

Or **atof** can be used in other contexts without being followed by the parentheses:

```
dispatch(atof);
```

In the second case, the name **atof** is converted to the address of that function, and the address is passed to the dispatch function.

4.2.3 Array References

Bracket operators are used to refer to elements of arrays. Given an array defined as array[10][5][2], you refer to a specific element within the array, as in the following example:

```
int array[10][5][2];
int i;
i = array[9][4][1];
```

This example assigns the value of the element in array[9][4][1] to i.

In addition, if an array reference is not fully qualified, it refers to the address of the first element in the dimension that is not specified. For example, the statements

```
int *ip;
ip = array[9][4];
```

assign the address of array[9][4][0] to the pointer ip. Therefore, you could write

```
ip = array[9];
```

to assign the address of array[9][0][0] to the pointer ip. Finally, a reference such as

```
ip = array;
```

assigns the address of the array's first element, array[0][0][0], to the pointer ip. A reference to an array name with no bracket operator is often used to pass the array's address to another function, as in:

```
funct(array);
```

Bracket operators can also be used to perform general address arithmetic of the form:

```
addr[intexp]
```

where addr is the address of some previously declared object (that is, a pointer-valued expression), and intexp is an integer-valued expression. The result of the expression is scaled, or multiplied, by the size in bytes of the addressed object; if intexp is a positive integer, the result is the address of a subsequent object of this size; if intexp is zero, the result is the address of the same object; if intexp is negative, the result is the address of a previous object.

4.2.4 Lvalues

The values of objects are sometimes categorized further, into lvalues and rvalues. The computer can be considered a machine that manipulates abstract objects which have a specific location and contain a specific value. The lvalue then denotes the location of an object. That location is used when the contents of the object are modified. The

rvalue denotes the contents of the object. It is used when the contents of the object are read. For instance, in the expression

```
x = y;
```

the contents of y (if y is a variable) are taken and assigned to x. In other words, the expressions use the rvalue of y and the lvalue of x in performing the assignment.

The following syntax defines those C expressions that either have or produce lvalues:[1]

```
lvalue ::=
        identifier
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
        * expression
        ( lvalue )
```

In order, these expressions represent:

1. Names of scalar variables, structures, and unions.
2. References to array elements (also scalars).
3. References to structure and union members (the meaning of the period operator or the right-arrow operator), except for references to fields which are not lvalues.
4. References to pointed-to objects (also called dereferenced pointers — that is, the asterisk followed by the name of a pointer variable or by another address-valued expression).
5. Any of the above, enclosed in parentheses.

All lvalue expressions represent a single location in the computer's memory.

4.2.5 Structure and Union References

A member of a structure or union can be referenced with either of two operators: the period or the right arrow.

A primary expression followed by a dot followed by an identifier refers to a member of a structure or union and is itself a primary expression. The first expression must be an lvalue naming a structure or union. The identifier must name a member of that structure or union. The result is a reference (if the member is a scalar) to the named member of the structure or union. The name of the desired member must be preceded

1. The word *lvalue* is sometimes used to mean either an lvalue or one of these expressions. The context usually makes the meaning clear. In this manual, lvalue means one of these expressions.

by a period-separated list of the names of all higher level members. For more information, see Section 3.4.2.

The other form for structure and union references uses the arrow operator. A primary expression followed by an arrow (built from a hyphen (-) and a greater-than symbol (>)) followed by an identifier refers to a member of a structure or union and is itself a primary expression. The first expression must be a pointer to a structure or a union. It must be some expression that results in a structure's or union's address. The identifier must name a member of that structure or union. The result is a reference to the named member.

4.3 Unary Expressions and Operators

Unary expressions are formed by combining a unary operator with a single operand. All unary operators are of equal precedence and group from right to left. They perform the following operations:

- Negate a variable arithmetically (-) or logically (!).
- Increment (++) and decrement (--) variables.
- Find addresses (&) and dereference pointers (*).
- Calculate a one's complement (~).
- Force the conversion of data from one type to another (cast).
- Calculate the sizes of specific variables or of types (**sizeof**).

4.3.1 Negating Arithmetic and Logical Expressions

The result of the expression

```
- expression
```

is the arithmetic negative of the expression. The usual arithmetic conversions are performed. The negative of an **unsigned** quantity is computed by subtracting its value from 2^{32}. There is no unary plus operator in C.

The result of the expression

```
! expression
```

is the logical (Boolean) negative of the expression. If the expression is zero, the result is 1; if the expression is not zero, the result is 0. The type of the result is **int**. The expression can be a pointer (or other address-valued expression) or an expression of any arithmetic type.

4.3.2 Incrementing and Decrementing Variables

The object referred to by the lvalue in the expression

```
++lvalue
```

is incremented before its value is used. The result is a new value of the object, not a reference to the object; for instance, the above expression could not appear by itself on the left side of an assignment expression.

The object referred to by the lvalue in the expression

```
lvalue++
```

is incremented after its value is used. The result is the value of the object before the increment, not a reference to the object.

If the operand is a pointer, the address it contains is incremented by the length of the addressed object.

The objects referred to by the lvalues in the expressions

```
--lvalue
lvalue--
```

are decremented analogously to the prefix and postfix ++ operators. Again, the result is the value of the object either before or after the decrement, not a reference to the object; that is, the expression --lvalue or lvalue-- cannot be used alone on the left side of an assignment expression.

If the operand is a pointer, the address it contains is decremented by the length of the addressed object.

4.3.3 Computing Addresses and Dereferencing Pointers

The expression

```
& lvalue
```

results in the address of the object to which the lvalue refers. The ampersand operator may not be applied to **register** variables or to bit fields in structures or unions.[1]

In the special case of argument lists, the ampersand operator may be applied to constants. This use of the ampersand operator passes constants to non-C functions that expect arguments to be passed by reference. This use is not recommended for other applications.

In the following

```
* expression
```

the asterisk operation means indirection. The expression must be a pointer or other address-valued expression, and the result is a reference to the object to which the expression points. The type of the addressed object is the type of the result.

1. In VAX-11 C, any **register** variable to which the ampersand operator is applied is simply changed to **auto**. No warning message is issued.

4.3.4 Calculating a One's Complement

The result of

```
~ expression
```

is the one's complement of the expression. The expression must be integral (an integer or character). The usual arithmetic conversions are performed.

4.3.5 Forcing Conversions to a Specific Type

The cast operator is used to force the conversion of an operand to a specified scalar data type. The operator consists of a data type name, written in parentheses, which precedes the operand expression:

```
(type-name) expression
```

The value of the expression is converted to the named type, just as if the expression were assigned to a variable of that type. If the operand is a variable, its value is taken and then converted to the named type. The variable's contents are not changed. The type name has the following formal syntax:

```
type-name ::=
        type-specifier abstract-declarator
```

In simple cases, the type specifier is the keyword for the data type, such as **char** or **double**. The type specifier may also be a struct-or-union specifier or an enum-specifier or a **typedef** tag.

The abstract declarator is a declaration without the identifier:

```
abstract-declarator ::=
        empty
        ( abstract-declarator )
        * abstract-declarator
        abstract-declarator ( )
        abstract-declarator [ constant-expression ]
```

To avoid confusion with the form

```
abstract-declarator( )
```

the abstract declarator may not be empty in:

```
(abstract-declarator)
```

Abstract declarators may include the brackets and parentheses that indicate arrays and function calls. However, cast operations may not force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in such operations as

```
(int (*)[]) P1
```

which casts P1 to "pointer to array of **int**." Note that this kind of cast operation in no way changes the contents of P1; it only causes the compiler to treat the value of P1 as a pointer to such an array. For

example, casting pointers this way can change the scaling that occurs when an integer is added to the pointer.

4.3.6 Calculating Sizes of Variables and Data Types

In the expressions

```
sizeof expression
sizeof (type-name)
```

the result is the size in bytes of the operand. In the first case, the result of **sizeof** is determined from the declarations of the objects in the expression. In the second case, the result is the size in bytes of an object of the named type. The syntax of type-name is the same as for the cast operator.

4.4 Binary Expressions and Operators

The binary operators fall into the following categories:

- Additive operators: addition (+) and subtraction (–).
- Multiplicative operators: multiplication (*), mod (%), and division (/).
- Equality operators: equality (==) and inequality (!=).
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).
- Bitwise operators: AND (&), OR (¦), and XOR (^).
- Logical operators: AND (&&) and OR (¦¦).
- Shift operators: left (<<) and right (>>).

4.4.1 Additive Operators

The additive operators + and – perform addition and subtraction. Their operands are transformed by the usual arithmetic conversions.

The address of an array element and a value of any integral type can be added. The compiler converts the integer to an address offset by multiplying the integer by the length of the addressed object. The result is the address of an object of the same type as the originally addressed object, where both objects are in the same array.

A value of any integral type may be subtracted from a pointer or address; then, the same conversions apply as for addition.

When two **enum** constants or variables are combined, the result is of type **int**.

If two addresses of objects of the same type are subtracted, the result is converted (it is divided by the length of the object) to an **int** representing the number of objects separating the addressed objects. The results of this conversion are unpredictable unless the two objects are in the same array.

4.4.2 Multiplicative Operators

The multiplicative operators *, /, and % perform the usual arithmetic conversions. The binary * operator performs multiplication. The binary / operator performs division. When integers are divided, truncation is toward zero.

The binary % (mod) operator divides the first operand by the second and yields the remainder. Both operands must be integral. The sign of the result is the same as the sign of the quotient. If b is not zero, then it is always true that (a/b)*b + a%b equals a.

4.4.3 Equality Operators

The equality operators == (equal to) and != (not equal to) perform the usual arithmetic conversions on their two operands. Like the relational operators, they produce a result of type **int**, so that

```
a<b == c<d
```

is 1 if both relational expressions have the same truth value, and 0 if they do not. Two pointers or addresses are equal if they identify the same storage location. A pointer or address can be compared with an integer, but the result is not portable unless the integer is zero; a null pointer is considered equal to zero.

Although different symbols are used for assignment and equality (= and ==, respectively), C allows either operator in some contexts, so you must be careful not to confuse them. For example:

```
if (x=1) statement-1;
else     statement-2;
```

always executes statement-1, since the value of the expression x=1 is 1.

4.4.4 Relational Operators

The relational operators compare two operands and produce a result of type **int**. The result is 0 if the relation is false and 1 if it is true. The operators are < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to). The usual arithmetic conversions are performed. If two pointers or addresses are compared, the result depends on the relative locations of the two addressed objects. Pointers to objects at lower addresses are "less than" pointers to objects at higher addresses. If two addresses indicate elements in the same array, the address of an element with a lower subscript is less than the address of an element with a higher subscript.

The operators group from left to right. However, note that the statement

```
if (a<b<c)...
```

compares c with the value 0 or 1; it does *not* mean "if b is between a and c...".

4.4.5 Bitwise Operators

These operators may be used only with integral operands (that is, with **char**s and with **int**s of all sizes). The usual arithmetic conversions are performed. The result is the bitwise AND (&), XOR (exclusive OR, ^), or OR (|) of the two operands. All operands are always evaluated.

4.4.6 Logical Operators

The logical operators are && (AND) and || (OR). These operators guarantee left-to-right evaluation. The right operand is not evaluated if the result is known from the evaluation of the left operand. The result (of type **int**) is either 0 or 1. That is:

```
E1 && E2
```

is 1 if both its operands are nonzero, or 0 if one operand is zero. If E1 is zero, E2 is not evaluated. Similarly:

```
E1 || E2
```

is 1 if either operand is nonzero, and 0 otherwise. If E1 is nonzero, E2 is not evaluated.

The operands of logical operators need not have the same type, but each must be one of the fundamental types or must be a pointer or other address-valued expression.

4.4.7 Shift Operators

The shift operators (<< and >>) take two operands, both of which must be integral. The usual arithmetic conversions are performed on both operands; then, the right-hand operand is converted to **int**, and the type of the result is the type of the left operand. The result of:

```
E1 << E2
```

is the value of expression E1 shifted to the left by E2 bits. Vacated bits are cleared. The result of

```
E1 >> E2
```

is the value of expression E1 shifted to the right by E2 bits. Vacated bits are cleared if E1 is **unsigned**; otherwise, the right shift is arithmetic — vacated bits are filled with a copy of E1's sign bit.

The result of the shift operation is undefined if the right-hand operand (E2) is negative or if the value of E2 is greater than 32 bits.

4.5 Conditional Expression and Operator

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. For example:

```
E1 ? E2 : E3
```

If E1 is nonzero, then E2 is evaluated. If E1 is zero, E3 is evaluated. Conditional expressions group from right to left. Conversions are made in the following order:

1. If possible, the usual arithmetic conversions are performed on E2 and E3, to make them have the same type.
2. Otherwise, if E2 and E3 are address expressions indicating objects of the same type, the result has that type.
3. Otherwise, either one of the E2 and E3 operands must be an address expression, and the other, the constant 0. The result has the type of the addressed object.

4.6 Assignment Expressions and Operators

In C, there are several assignment operators. An "assignment" is not only an operation but also an expression. Assignments result in the value of the target variable after the assignment. They can be used as subexpressions in larger expressions.

The set of assignment operators consists of the equal sign (=) alone and in combination with binary operators. An assignment expression has two operands — an lvalue and an expression — separated by one of these operators. An assignment expression such as

E1 *op*= E2

is equivalent to

E1 = E1 *op* (E2)

For example, the expression

```
E1 += E2;
```

is equivalent to

```
E1 = E1 + E2;
```

E1 is evaluated only once and must be an lvalue. The type of the assignment expression is the type of E1, and the result is the value of E1 after the operation is performed. E2 is parenthesized above because it could contain other operators that are of lower precedence than op. For example, the expression

```
a += b << 1;
```

is the same as

```
a = a + (b << 1);
```

not

```
a = a + b << 1;
```

In the simple assignment expression

```
E1 = E2
```

the value of E2 replaces the previous value in E1. Another example, the expression

```
array[1] += 100;
```

adds 100 to the contents of array[1]. The result of the expression is the result of the addition and has the same type as array[1].

If both assignment operands are arithmetic, the right operand is converted to the type of the left before the assignment is made.

The simple assignment operator (=) can be used to assign structures and unions. All other assignment operators, all right operands, and all left operands that are not pointers must be arithmetic. If the operator is -= or +=, the left operand may be a pointer, and the right operand (which must be integral) is converted in the same manner as that in which the binary + and - operators are converted.

An address may be assigned to an integer, an integer to a pointer, and the address of an object of one type to a pointer of another type. Such assignments are simple copy operations, with no conversions. This usage may cause addressing exceptions when the resulting pointers are used. However, if the constant 0 is assigned to a pointer, the result is a null pointer. The null pointer is distinguishable (by the equality operators) from a pointer that points to any object.

For the sake of compatibility with older implementations of C, VAX-11 C allows certain deviations from the spellings of compound assignment operators shown in Table 4-1. Namely:

- When the operators are written in the order shown in Table 4-1, the two characters can be separated by white space. That is:

  ```
  E1 += E2;
  ```

  and

  ```
  E1 + = E2;
  ```

  are identical.

- The operators can also be written with the characters in reverse order, as in:

  ```
  E1 =+ E2;
  ```

However, you should avoid the second form for the following reasons:

- The syntax allowed by VAX-11 C is more restrictive in this case. Specifically, the characters *, +, -, and &, because they also appear in unary operators, must be immediately adjacent to the '=' character in this form. This placement avoids ambiguities in such cases as:

  ```
  E1 =*p;
  ```

  which multiplies the value in E1 by the value of p.

- Even with usage that follows the guidelines, it is possible to introduce ambiguities, as in:

  ```
  E1 =/*part of a comment...
  ```

4.7 Comma Expression and Operator

When two expressions are separated by the comma operator, they are evaluated from left to right, and the result of the left expression is discarded. For example:

```
R = (T = 1, T += 2);
```

assigns 3 to both R and T.

The type and value of the result of a comma expression are the type and value of the right operand. The operator groups from left to right.

Note that comma expressions must be parenthesized if they appear where commas have some other meaning, as in argument and initializer lists. For example:

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

calls the function f with the arguments a, 5, and c. In addition, t is assigned the value 3.

Chapter 5

Statements

This chapter describes the statements in the C programming language. Except as indicated, statements are executed in the sequence in which they appear in a program.

5.1 Expression Statement

You can use any valid expression as a statement by terminating it with a semicolon:

expression ;

5.2 Compound Statement

A compound statement in C is sometimes called a block and allows more than one statement to appear where a single statement is required by the language. A compound statement has the form:

compound-statement ::=
{ *declaration-list statement-list* }

That is, the block is an optional list of declarations followed by a list of statements, all enclosed in braces. If you include declarations, the variables they declare are local to the block and, for the rest of the block, they supersede any previous declaration of variables of the same name. Variables of storage class **auto**, **register**, or **static** can have initializers in the declaration list.

A block is entered "normally" when control flows into it, or when a **goto** statement transfers control to the label of the block itself. Any **auto** or **register** variables are initialized each time the block is entered normally; the initializations do not occur if a **goto** statement refers to a label inside the block or if the block is the body of a **switch** statement.

5.3 if Statement

A conditional **if** statement can be written with or without an **else** clause:

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

In each case, the expression is evaluated, and if it is not zero, the first statement is executed.

Note that all relational operators define a "true" result to be nonzero so that the expression in any **if** statement (or any other conditional statement) may be a relational expression with predictable results (nonzero or zero).

If the **else** clause is included and the expression is zero, the **else** statement is executed. In a series of **if-else** clauses, the **else** matches the most recent **else**-less if.

5.4 while Statement

The **while** statement has the form:

while (expression) statement

The expression is evaluated before each execution, and the statement is executed zero or more times, as long as the expression is not zero.

5.5 do Statement

The **do** statement has the form:

do statement **while** (expression) ;

The statement is executed at least once, and the expression is evaluated after each execution. If the expression is not zero, the statement is executed again.

5.6 for Statement

The **for** statement has the form:

for (*expression-1* ; *expression-2* ; *expression-3*)
statement

The **for** statement executes a statement zero or more times. It uses three control expressions, as shown. (Note that expression-3 is not followed by a semicolon.) A **for** loop is executed in the following steps:

1. Expression–1 is evaluated before the first iteration of the loop, and only once. It usually specifies the initial values for variables.

2. Expression–2 is a relational or logical expression that determines whether to terminate the loop. Expression–2 is evaluated before each iteration. If it is zero, execution of the **for** statement terminates. If it is not zero, the statement is executed.
3. Expression–3 is evaluated after each iteration. It usually specifies increments for the variables initialized by expression–1.
4. Iterations of the **for** statement continue until expression–2 produces a "false" (zero) value, or until some statement, such as **break** or **goto**, interrupts it.

The **for** statement is equivalent to:

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

The VAX–11 C compiler optimizes certain **for** statements for simple loops such as

```
for(i=0; i<15; i++)
        printf("%d\n",i);
```

So the use of **for** statements rather than the equivalent **while** statement is preferred.

Any of the three expressions in a loop may be omitted. If expression–2 is omitted, the test condition is always true; that is, the **while** in the above expansion becomes **while**(1). If either expression–1 or expression–3 is omitted from the **for** statement, that expression is effectively dropped from the above expansion.

The construct

```
for (;;) statement
```

is an infinite loop. It can be terminated by a **break**, **return**, or **goto** within the statement.

5.7 break Statement

The **break** statement has the form:

```
break ;
```

It terminates the immediately enclosing **while**, **do**, **for**, or **switch** statement. Control passes to the statement following the terminated statement.

5.8 switch Statement

The **switch** statement has the form:

```
switch ( expression ) statement
```

The **switch** statement executes one or more of a series of cases, based on the value of the expression.

The usual arithmetic conversions are performed on the expression, but the result must be **int**. The statement is typically a compound statement, within which any statement or list of statements can be prefixed with one or more **case** labels:

case constant-expression :

where the constant expression must also be **int**. No two **case** labels may specify the same value. The value of any constant expression must be between -32,768 and 32,767.

At most one statement in the compound statement may have the label:

default :

The **case** and **default** labels may occur in any order. When the **switch** statement is executed, the following sequence takes place (note that each case "flows into" the next unless explicit action is taken, such as a **break** statement):

1. The **switch** expression is evaluated and compared with the constant expressions in the **case** labels.
2. If a **case** label matches the expression's value, the statement or list of statements following that label is executed. If the list of statements ends with the **break** statement, the **break** terminates the **switch** statement; otherwise, the next case encountered is executed. (See Example 5-1.) The **switch** statement can also be terminated by a **return** or **goto** statement; if the **switch** is inside a loop, it can be terminated by a **continue** statement.
3. If no **case** label matches the expression's value, but there is a **default** case, the **default** case is executed. It need not be the last case listed. If a **break** statement does not end the **default** case, the next case encountered is executed.
4. If there is no **case** for the expression's value and there is no **default**, the body of the **switch** statement is not executed.

NOTE

If declarations appear in the compound statement within a **switch** statement, any initializations of **auto** or **register** variables are ineffective. However, this rule does not apply to compound statements following a **case** label.

In general, the **break** statement must be used to ensure that a **switch** executes as expected. Example 5-1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal. A series of **case** statements is used to increment the counters. The **break** statement causes the program to return to the beginning of the **while** loop when one of the counters is incremented. The program returns automatically to the beginning of the **while** loop if none of the counters is incremented.

```
#include stdio

/* A PROGRAM TO COUNT BLANKS, TABS, AND NEWLINES. */
main()
{
     int number_tabs = 0;
     int number_lines = 0;
     int number_blanks = 0;
     int ch;
     while ((ch = getchar()) != EOF)
          switch (ch) {
                         case '\t': ++number_tabs; break;
                         case '\n': ++number_lines; break;
                         case ' ': ++number_blanks; break;
                         }

     printf("Blanks\tTabs\tNewlines\n");
     printf("%6d\t%6d\t%6d\n",number_blanks,
          number_tabs,number_lines);

}
```

Example 5–1: Use of switch to Count Blanks, Tabs, and Newlines

The program responds to the following input:

```
Every good boy, RET
The quick brown fox, RET
Line with 2 TAB TAB tabs, RET
^Z
```

by writing out:

```
Blanks        Tabs  Newlines
     7           2           3
```

On the other hand, if the **break** statements were omitted, the program would write out:

```
Blanks        Tabs  Newlines
    12           2           5
```

Without the **break** statements, each case drops through to the next case. The number shown for tabs happens to be right, because the tabs case is first in the **switch** statement and is executed only if ch == '\t'. Notice that the number shown for newlines is the correct number plus the number of tabs, and the number shown for blanks is the total of all three cases.

5.9 continue Statement

The **continue** statement has the form:

continue ;

The **continue** statement immediately passes control to the bottom of the immediately enclosing **while**, **do**, or **for** statement.

In each of the following statements, a **continue** is equivalent to **goto label**:

```
while (...) { ... label: ; }

do { ... label: ; } while (...);

for (...; ...; ...) { ... label: ; }
```

continue is intended only for loops, not for **switch**es. A **continue** inside a **switch** inside a loop causes reiteration of the enclosing loop.

5.10 return Statement

The **return** statement has the form:

return *expression* ;

The **return** statement causes a return from a function, with or without a return value.

The return value is undefined unless specified in a **return** statement. When an expression is specified in the **return** statement, it is evaluated and the value is returned to the calling function; the value is converted, if necessary, to the type with which the called function was declared.

A function that does not have a **return** statement is the same as a **return** statement that does not specify an expression; it does not return a value to the calling function.

5.11 goto Statement

The **goto** statement has the form:

goto identifier ;

The **goto** statement transfers control unconditionally to the labeled statement. The identifier must be a label located in the current function.

goto may be used to branch into a block, but any automatic variables declared in the block will not be initialized.

5.12 Labeled Statement

A label has the form:

identifier:

Any statement can be preceded by a label. The scope of a label is the current function. Since the label name is independent of the scope rules applied to variables, there can be variables with the same name as the label in the function that contains the label. Labels are used only as the targets of **goto** statements.

5.13 Null Statement

A null statement is a semicolon:

;

Null statements are used to provide null operations in situations where the grammar of the language requires a statement. In particular, the bodies of **if-else**, **while**, **do**, and **for** statements are not optional, so the null statement is often used to write these statements with null bodies. The most common use is in loop operations where all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first zero element of an array known to have a zero element:

```
for(i=0; array[i] != 0; i++) ;
```

Chapter 6

Library Functions

Because the C language has no predefined or built-in functions, all C compilers are supplied with libraries of common functions that can be used by C programmers on their particular system. This chapter describes the library functions supplied with the VAX-11 C compiler. The function descriptions are grouped alphabetically in Tables 6–1 through 6–8, according to the following categories:

- Input/output (I/O) functions
- Character classification functions
- String-handling functions
- Character conversion functions
- Mathematical functions
- Memory allocation functions
- Miscellaneous functions
- UNIX emulation functions

Section 6.11 describes each function in detail.

Programs that use these library functions must contain a C function named main or a C function with the main__program attribute.

NOTE

When an error occurs during a call to any of the functions described in this chapter, the function returns an unsuccessful status and sets the external variable **errno** to a value which indicates the reason for the failure. See Appendix E for more information.

6.1 Performing I/O from C Programs

The C programming language has no built-in I/O functions. However, each implementation usually provides some I/O capability in the form of library functions. The amount of I/O support varies from implementation to implementation; the user interface and language functionality also differ between implementations.

As shown in Figure 6–1, VAX-11 C makes available four distinct hierarchical levels of I/O. The lowest level, VAX/VMS system services, is

closest to the operating system; the highest level, standard I/O, is farthest. For the duration of a program, each level of access is exclusive of the others; you cannot access a file from different levels in the same program.[1]

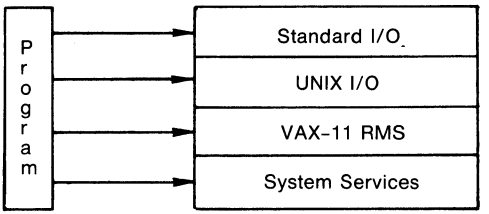


Figure 6-1: I/O Interface from C Programs

Before deciding which level is appropriate for you, you must first ask the question: Are you concerned with UNIX compatibility, or are you developing code that will run solely under VAX-11 C? If UNIX compatibility is important to you, you will probably want to use the two highest levels of I/O — standard I/O and UNIX I/O — because they are more independent of the operating system. The two highest levels are also easier to learn quickly, an important consideration for new programmers.

If UNIX compatibility is not important to you, or you require the sophisticated file processing that the standard I/O and UNIX I/O levels do not provide, then you will find VAX-11 RMS desirable. Note that the use of RMS is mandatory for RMS relative and indexed file organizations. For a description of RMS, see Chapter 8.

If you are writing system-level software, you may need to directly access VAX/VMS through calls to system services. For example, you may need to directly access a user-written device driver through QIO$ (Queue I/O) system service requests. To do this, you would need to use the VAX/VMS level of I/O; this level is recommended for experienced VAX/VMS programmers only. Chapter 9 contains some examples of programs that call VAX/VMS system services.

The UNIX and standard I/O functions are contained in the VAX-11 C run-time library. Only those UNIX and standard I/O functions that are appropriate for VAX/VMS are provided. That is, some functions that may be provided by other implementations are not provided by VAX-11 C because those functions conflict with VAX/VMS. In some cases, conflicting functions are replaced by an equivalent VAX-11 C function. For example, the UNIX **unlink** function has been replaced by

---

1. The exception to this rule is the **fdopen** function. It allows a file opened by UNIX I/O functions to be accessed by standard I/O functions. However, after an **fdopen** function call, you can use only standard I/O functions to access the file.

the VAX-11 C **delete** function. Table 6-1 lists the run-time functions that perform I/O on the UNIX and standard levels.

As shown in Figure 6-2, the UNIX and standard I/O functions map to RMS and, subsequently, to VAX/VMS. When you create a file on either of these levels, you are actually creating an RMS sequential file. The standard I/O functions create sequential files with stream record format. The UNIX I/O functions create stream files by default, but you may specify certain record attributes (including the record format) when you create the file.

6.1.1 Stream Files and Stream Access

Stream files are files treated as streams of bytes. A series of bytes is read from or written to a stream file directly, with no record structure or implied carriage control. In VAX-11 C, and in most other implementations of C, stream files and their associated functions form the standard I/O facilities.

Stream files are created by the **fopen** and **create** functions. For example, a call to **fopen** that opens a new file for output creates a stream file. A call to **fopen** that opens an existing file for input presents the program with a stream file that is processed with the conventional stream I/O functions, such as **fseek**, **ftell**, **fread**, **fwrite**, and **fprintf**.

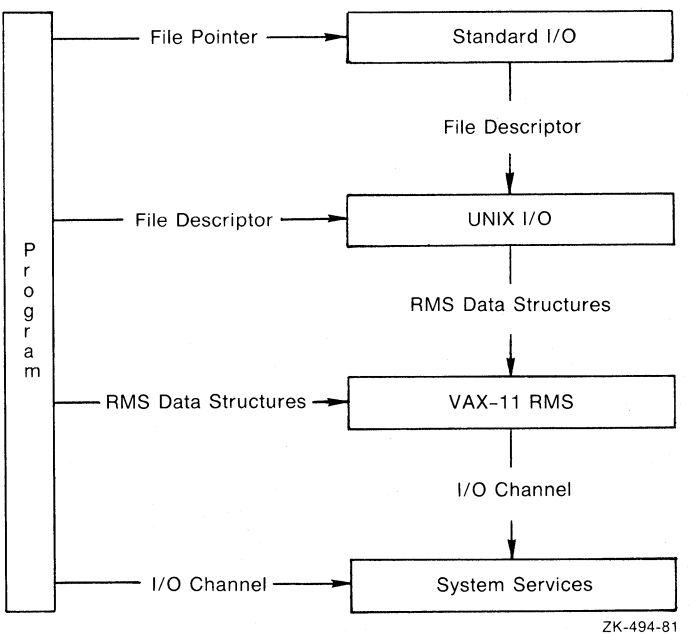


Figure 6-2: Mapping Standard and UNIX I/O to RMS

6.1.1.1 Relationship to VAX-11 Record Management Services (RMS)

Stream files in VAX-11 C correspond to VAX-11 RMS stream files with the line feed terminator attribute. VAX-11 C permits stream access to stream files. It also permits stream I/O operations on RMS record files, but the positioning options of stream access are more restricted with record files.

The next sections review the stream access options permissible with stream files and define the extent to which stream access is permitted with RMS record files.

If you are not familiar with RMS, you should consult the manuals listed at the beginning of Chapter 8 before continuing with this section.

6.1.1.2 Stream Access to Stream Files

Stream access to stream files uses the stream I/O facilities of RMS. A stream of bytes is either written to or read from a file with no translation. If the file has been opened for update, it can be read (**fread**) and written (**fwrite**) at the current byte position in the file.

An **fread** followed by an **fwrite** places bytes in the file after the last byte of the previous **fread**. An **fwrite** followed by an **fread** causes reading to begin after the last byte of the previous **fwrite**.

A stream file can be positioned to an arbitrary byte at any time (**fseek**). If positioned beyond the end-of-file, then the file is extended with zero bytes. The file may be positioned relative to the beginning-of-file, relative to the current position, or relative to the end-of-file. The first byte in the file is byte zero; therefore, specifying zero as the absolute position in an **fseek** call positions the file at its first byte. You can also determine the current byte position of a stream file with the **ftell** function.

You must open a file for update if the file is going to be written randomly. For example:

```
#include stdio

main()
{
     FILE *outfile;
     outfile = fopen("diskfile.dat","w+");
     .
     .
}
```

Here, the stream file diskfile.dat is opened for "write update" access. (For the distinction between "read update" and "write update," see **fopen**, Section 6.11.49.)

6.1.1.3 Stream Access to Record Files

Stream access to record files is done with the record I/O facilities of RMS. A byte stream is emulated by translating carriage control during the process of reading and writing records. Random access is allowed to record files, but positioning (with **fseek**) must be on a record boundary,

and writes followed by reads (or reads followed by writes) do not work as with stream files. Positioning of a record file causes all buffered input to be discarded and buffered output to be written to the file.

Stream input from record files is emulated by the run-time support in two steps. First, a logical record is read from the file. Second, the record is expanded to simulate a stream of bytes by translating the record's carriage-control information (if any). In RMS terms, the translation is performed by one of the following steps:

- If the record attributes are implied carriage control (RAT=CR), then a newline is appended to the record.
- If the record attributes are print carriage control (RAT=PRN), then the prefix and postfix carriage controls are expanded and concatenated before and after the record.
- If the record attributes are FORTRAN carriage control (RAT=FTN), then the first byte of the record is removed, and prefix and postfix characters are concatenated to the record. The following rules describe the way the character in the first byte maps onto the prefix and postfix bytes that appear in the emulated stream. <record> denotes the bytes contained in the logical record exclusive of the first, carriage-control byte; '\n' denotes the newline character; '\f' denotes the form-feed character; '\r' denotes the carriage-return character:

  NUL → <record>

  0 → \n<record>\n

  1 → \f<record>\n

  + → <record>\r

  $ → \n<record>

  all others → <record>\n

- If the input is coming from a nonterminal file, then the record is passed unchanged to the user program with no prefix or postfix characters added to it.

  If the record attributes are null (RAT=null) and the input is coming from a terminal, then the terminator is appended to the record. If the terminator is a carriage return or CTRL/Z, then it is translated to a newline.

- If the record format is variable length with fixed control (RFM=VFC), and the record attributes are not print carriage control (RAT *is not* PRN), then the fixed-control area is concatenated to the front of the record.

As you read from the file, the VAX-11 C run-time support delivers a stream of bytes resulting from the above translations. Information that is not read from an expanded record by one function call is delivered on the next input function call.

CAUTION

An expanded record cannot exceed 512 bytes. Thus, the input record generally must not exceed 510 bytes of actual data, since up to two characters may be added in the expansion process.

Stream output to record files is performed by the VAX-11 C run-time support in two steps. First, a logical record is formed from the bytes specified by the output function (**fwrite**, for example) by translating any carriage-control bytes into RMS terms. Then, the logical record is written out.

The first part of the stream output emulation is the formation of a logical record. As you write bytes to a record file, the emulator examines the information being written for record boundaries. The handling of information in the byte stream depends on the attributes of the destination file or device, as follows:

- If the record attributes specify no carriage-control information (RAT=null), then the stream of bytes presented in an output-function call is taken to be a logical record.
- If the destination file or device being written to has carriage-control information (RAT=CR, RAT=FTN, or RAT=PRN), then the emulator buffers output bytes while it searches for a newline character. Up to 512 bytes are buffered. If more than 512 bytes are encountered before a newline is encountered, then an error is signaled and the buffer is written out. Otherwise, when a newline is found, the logical record is formed by appending the newline to the buffered bytes.

The second part of stream output emulation is the actual writing of the logical record formed during the first step. One of the following steps is executed to form the output record:

- If the output file record format is variable length with fixed control (RFM=VFC), and the record attributes do not include print carriage control (RAT *is not* PRN), then the beginning of the logical record is taken to be the fixed-control header, and the number of bytes written out is reduced by the length of the header. If there are too few bytes in the logical record, an error is signaled.
- If the record attribute is carriage control (RAT=CR), and if the logical record ends with a newline, the newline is dropped, and the logical record is written out with implied carriage control.
- If the record attribute is print carriage control (RAT=PRN), then the record is written with print carriage control. If the logical record ends with a newline, the newline is dropped, and the output record is preceded by a line feed and followed by a carriage return.
- If the record attribute is FORTRAN carriage control (RAT=FTN), then the logical record is written out with FORTRAN carriage control. If the logical record ends with a newline, the newline is

dropped and a space character is inserted at the front of the record. Otherwise, a NUL is inserted at the front of the record.

- If the record attribute is null (RAT=null), then a test is performed to determine whether the logical record is being written to a terminal device. If so, the record is scanned, and each newline that is encountered is replaced by a carriage-return/line-feed pair. The record is then written out with no carriage control.

6.1.2 Standard I/O

The standard I/O functions access the file by a file pointer. A file pointer points to a file control block, which is defined in the stdio **#include** module as a preprocessor substitution (similar to a **typedef**) for a data type named FILE. The structure contains the definition of an RMS sequential file with stream record format.

A file pointer is declared as follows:

```
FILE *infile;
```

In this case, infile is a pointer to a FILE structure.

The VAX-11 C run-time library contains the following functions that access files by file pointer:

fopen	**fseek**	**fgets**	**fputs**
fclose	**ftell**	**fscanf**	**ungetc**
fileno	**rewind**	**fwrite**	**fgetname**
freopen	**fread**	**fprintf**	**clearerr**
setbuf	**getc**	**putc**	**ferror**
feof	**fgetc**	**fputc**	**putw**
fflush	**getw**		

6.1.3 UNIX I/O

The UNIX I/O functions access the file by a file descriptor. A file descriptor is an integer that identifies the file. A file descriptor is declared as follows:

```
int fd;
```

In this case, fd is the name of the file descriptor.

When you create a file on the UNIX I/O level, you can supply values for the following RMS file attributes:

- Allocation quantity
- Block size
- Default file extension
- Default file name
- A number of file-processing options
- Multiblock count
- Multibuffer count
- Maximum record size
- Record attributes
- Record format

Functions such as **creat** associate the file descriptor with a file. For example:

```
fd = creat("infile",0,"rat=cr","rfm=var");
```

This statement creates the file infile., with mode argument 0, carriage-return control, and variable-length records, and it associates the file descriptor, fd, with the file. When the file is accessed for other operations, such as reading or writing, the file descriptor is used to refer to the file. For example:

```
write(fd,buffer,sizeof(buffer));
```

This statement writes the contents of the buffer to infile.

The VAX-11 C run-time library contains the following functions that access files by file descriptor:

creat	**close**	**dup**	**dup2**
open	**pipe**	**lseek**	**read**
write	**getname**		

6.1.4 Predefined Files

VAX-11 C defines three file pointers that perform I/O to and from the logical devices usually associated with the user's terminal (for interactive jobs) or a batch stream (for batch jobs).[1] These file pointers are defined when you include the stdio module with the **#include** preprocessor control line.

The file pointer **stdin** is associated with the terminal to perform input. This file is equivalent to SYS$INPUT. The file pointer **stdout** is associated with the terminal to perform output. This file is equivalent to SYS$OUTPUT. The file pointer, **stderr**, is associated with the terminal to report run-time errors. This file is equivalent to SYS$ERROR.

Three file descriptors also exist that refer to the terminal. The file descriptor 0 is equivalent to SYS$INPUT, 1 is equivalent to SYS$OUTPUT, and 2 is equivalent to SYS$ERROR.

When performing I/O at the terminal, you can use standard I/O functions (giving the name **stdin**, **stdout**, or **stderr** as an argument), you can use UNIX I/O functions (giving the file descriptor as an argument), or you can use the following functions, which specifically perform I/O at the terminal:

getchar
putchar
gets
puts
scanf
printf

1. Since the three process permanent files SYS$INPUT, SYS$OUTPUT, and SYS$ERROR perform the same functions for both interactive and batch jobs, the term *terminal I/O* refers to both terminal and batch stream I/O.

Table 6-1: Input/Output Functions

Name	Description	#include Module	Page
Opening and Closing Files			
close	Closes a file	—	103
creat	Creates a new file	—	104
dup, dup2	Creates new descriptors for existing files	—	108
fclose	Closes a file	stdio	113
fdopen	Creates a FILE structure and associates it with a file descriptor	stdio	113
fileno	Returns an integer file descriptor	stdio	115
fopen	Opens a file	stdio	116
freopen	Reassigns the address of a FILE structure and opens the file	stdio	118
open	Opens a file for reading, writing, or both	—	132
pipe	Allows two processes to share data with read and write calls	—	133
setbuf	Associates a buffer with an input or output file	stdio	144
tmpfile	Creates a temporary file for use during a process; the file is deleted when the process (and its forks) is terminated	stdio	158
Positioning in Files			
feof	Tests for end-of-file	stdio	114
fflush	Writes out any buffered information to the file	stdio	115
fseek	Places you at a specified byte offset relative to the beginning of the file, the end of the file, or the current location within the file	stdio	119
ftell	Returns the current byte offset from the beginning of the file to the current location within the file	stdio	119

Table 6-1: (Cont.) Input/Output Functions

Name	Description	#include Module	Page
lseek	Places you at a byte offset within a file and returns the new position as an integer	—	130
rewind	Places you at the beginning of the file	stdio	141
Input Functions			
fread	Reads a specified number of items from the file	stdio	117
fgetc	Returns the next character from a file; generates a true function call	stdio	120
fgets	Reads a line from a file; the line is terminated by a NUL character	stdio	123
fscanf	Performs formatted input from a file	stdio	141
getc	Returns the next character from a file; implemented as a macro	stdio	120
getchar	Returns the next character from the standard input device	stdio	120
gets	Reads a line from the standard input device; the newline is replaced with a NUL character	stdio	123
isatty	Determines if a file descriptor is associated with a terminal	—	125
read	Reads a specified numbers of bytes from a file and places them in a buffer	—	140
scanf	Performs formatted input from the standard input device	stdio	141
sscanf	Performs formatted input from memory	stdio	141
Output Functions			
delete	Deletes a file	—	108
fgetname	Returns the file specification for a given file pointer	stdio	122
fprintf	Performs formatted output to a specified file	stdio	134
fputc	Writes a single character to a file; generates a true function call	stdio	139

Table 6–1: (Cont.) Input/Output Functions

Name	Description	#include Module	Page
fputs	Writes a string to a file	stdio	139
fwrite	Writes the specified number of items to the file	stdio	120
getname	Returns the file specification for a given file descriptor	stdio	122
printf	Performs formatted output to the standard output device	stdio	134
putc	Writes a single character to a file; implemented as a macro	stdio	139
putchar	Writes a single character to the standard output device	stdio	139
puts	Writes a string to the standard output device; terminates the string with a newline	stdio	139
putw	Writes a specified integer to a file	stdio	139
sprintf	Performs formatted output to a character string in memory	stdio	134
ungetc	Writes a character to a file buffer and leaves the file positioned before the character	stdio	160
write	Writes a number of bytes from a buffer to a file	—	162
Error-Handling Functions			
clearerr	Resets the error and end-of-file indicators	stdio	103
ferror	Returns a nonzero integer if an error occurs during read or write operations	stdio	115

6.2 Character Classification

The functions in Table 6–2 operate on characters. All of the functions in this table take a single argument and perform a logical operation. The argument can have any value. In the case of **isascii**, the function returns a logical result which states whether the argument is an ASCII

character (0 to 177 octal). The other functions return a logical result which states whether the argument is a particular type of ASCII character.

Appendix G contains a table of the ASCII character set. For each ASCII character, the table shows which character classification functions return a true value.

Table 6–2: Character Classification Functions

Name	Description	#include Module	Page
isalnum	Determines if the argument is alphanumeric	ctype	125
isalpha	Determines if the argument is alphabetic	ctype	125
isascii	Determines if the argument is an ASCII character	ctype	125
iscntrl	Determines if the argument is a control character	ctype	126
isdigit	Determines if the argument is a digit	ctype	126
isgraph	Determines if the argument is a graphic character	ctype	126
islower	Determines if the argument is a lowercase letter	ctype	126
isprint	Determines if the argument is a printing character	ctype	127
ispunct	Determines if the argument is a punctuation character	ctype	127
isspace	Determines if the argument is a space, horizontal or vertical tab, carriage return, form feed, or newline	ctype	127
isupper	Determines if the argument is an uppercase letter	ctype	127
isxdigit	Determines if the argument is a hexadecimal digit	ctype	128

6.3 String Handling

The functions in Table 6–3 manipulate strings. Some concatenate strings. Others search strings for specific characters or perform other lexicographic comparisons, such as determining the equality of two strings.

Table 6–3: String-Handling Functions

Name	Description	#include Module	Page
strcat	Concatenates two strings	—	152
strchr	Searches a string for the first occurrence of a given character	—	153
strcmp	Performs lexicographic comparison of two ASCII strings	—	153
strcpy	Copies one string to another	—	154
strcspn	Searches a string for a character within a set and returns the number of characters preceding the first match	—	154
strlen	Returns the length of a string	—	155
strncat	Concatenates two strings up to a maximum number of characters	—	152
strncmp	Performs lexicographic comparison of two ASCII strings (up to a maximum number of characters)	—	153
strncpy	Copies a maximum number of characters from one string to another	—	154
strpbrk	Searches a string for a character within a set and returns the address of the first match	—	156
strrchr	Searches a string for the last occurrence of a given character	—	153
strspn	Searches a string for the first occurrence of a character that is not in the search set	—	156

6.4 Character Conversion

The functions in Table 6–4 perform character and arithmetic conversions.

Table 6–4: Character Conversion Functions

Name	Description	#include Module	Page
atof	Converts an ASCII string to a numeric value (**double**)	math	100
atoi	Converts an ASCII string to a numeric value (**int**)	—	100
atol	Converts an ASCII string to a numeric value (**long**)	—	100
ecvt	Converts a **double** value to a NUL-terminated ASCII string	—	108
fctv	Converts a **double** value to a NUL-terminated ASCII string	—	108
gcvt	Converts a **double** value to a NUL-terminated ASCII string of digits	—	108
toascii	Converts an 8-bit ASCII character to a 7-bit ASCII character	—	159
tolower **__tolower**	Converts uppercase characters to lowercase; returns lowercase characters unchanged	—	159
toupper **__toupper**	Converts lowercase characters to uppercase; returns uppercase characters unchanged	—	159

6.5 Mathematical Functions

Table 6–5 shows the library functions that perform mathematical operations.

The errno definition file defines two run-time error return values for mathematical functions. EDOM causes an error message to be written to **stderr** when an argument is inappropriate; that is, when the argument is not within the function's domain. ERANGE causes an error message when a result is out of range; that is, when the argument is too large to be represented by the machine.

Table 6–5: Mathematical Functions

Name	Description	#include Module	Page
abs	Returns the absolute value of the integer argument	—	98
acos	Returns a value in the range 0 to pi which is the arc cosine of the radian argument	—	98
asin	Returns a value in the range -pi/2 to pi/2 which is the arc sine of the radian argument	math	99
atan	Returns a value in the range -pi/2 to pi/2 which is the arc tangent of the radian argument	math	99
atan2	Returns a value in the range -pi to pi which is the arc tangent of the two arguments	math	100
cabs	Returns "sqrt(x*x + y*y)"	math	124
ceil	Returns the smallest value which is equal to or greater than the argument	math	101
cos	Returns the cosine of the radian argument	math	103
cosh	Returns the hyperbolic cosine of the argument	math	104
exp	Returns the base e raised to the power of the argument	math	112
fabs	Returns the absolute value of the floating-point argument	math	98
floor	Returns the largest integer which is less than or equal to the argument	math	116

Table 6-5: (Cont.) Mathematical Functions

Name	Description	#include Module	Page
frexp	Returns the mantissa of the argument	math	119
hypot	Returns "sqrt(x*x + y*y)"	math	124
ldexp	Returns the first argument times 2 to the power of the second argument	math	128
log	Returns the natural logarithm of the argument	math	129
log10	Returns the base 10 logarithm of the argument	math	129
modf	Returns the fractional part and the integral part of the argument	math	131
pow	Returns the first argument raised to the power of the second argument	math	134
rand	Returns pseudorandom numbers	—	140
sin	Returns a value that is the sine of the radian argument	math	150
sinh	Returns a value that is the hyperbolic sine of the argument	math	151
sqrt	Returns the square root of the argument	math	151
srand	Reinitializes the random-number generator	—	140
tan	Returns the tangent of the radian argument	math	157
tanh	Returns the hyperbolic tangent of the argument	math	157

6.6 Memory Allocation

The functions in Table 6–6 allow you to control the allocation of memory from a C program. The functions **calloc**, **malloc**, and **realloc** return the address of the allocated area. They return a null pointer if there was insufficient memory.

The memory allocation functions in this section and the facilities listed below are mutually exclusive and should not be used in the same program:

- The functions **brk** and **sbrk**
- The VAX/VMS system services $EXPREG and $CNTREG
- The VAX-11 Common Run-Time Procedure Library functions LIB$GETVM and LIB$FREEVM

Table 6–6: Memory Allocation Functions

Name	Description	#include Module	Page
calloc	Allocates and clears an area of memory	—	101
cfree	Deallocates the space allocated by **calloc** or **realloc**	—	118
free	Deallocates the space allocated by **malloc** or **realloc**	—	118
malloc	Allocates the specified number of contiguous bytes of memory	—	131
realloc	Changes the size of an area previously allocated by **calloc** or **malloc**	—	141

6.7 Miscellaneous Functions

The functions in Table 6–7 perform miscellaneous services, such as identifying the process's user or terminal, or setting and generating signals.

Table 6–7: Miscellaneous Functions

Name	Description	#include Module	Page
ctermid	Returns the name of the controlling terminal	stdio	106
cuserid	Returns the name of the user who initiated the controlling process	stdio	107
gsignal	Raises a specified software signal	signal	124
longjmp	Returns to the context saved by **setjmp**	setjmp	145
mktemp	Creates a file name from a template	—	131
perror	Writes (to **stderr**) the most recent error encountered by VAX/VMS during execution of a C program	—	133
setjmp	Saves the context of the calling function for a subsequent **longjmp** call	setjmp	145
signal	Establishes the action to be taken when a specific signal is raised	signal	147
sleep	Suspends the current process for at least the specified number of seconds	—	151
ssignal	Establishes the action to be taken when a specific signal is raised	signal	152
tmpnam	Creates a character string to take the place of the file-name argument of other function calls	stdio	158

6.8 UNIX Emulation

The functions in Table 6–8 emulate UNIX/C functions of the same name. The emulation functions can help you convert C programs written for the UNIX system to C programs that will run on a VAX/VMS system.

Table 6–8: UNIX Emulation Functions

Name	Description	#include Module	Page
abort	Terminates the process	—	98
access	Checks a file for a specific access mode	—	98
alarm	Sends a signal to the process after a specified number of seconds	—	99
brk	Returns the lowest virtual address that is not used by the program	—	101
chdir	Changes the default directory	—	102
chmod	Changes the protection of the named file	—	102
chown	Changes the owner user identification code of the file	—	103
ctime	Converts the current time to an ASCII string	time	106
execl	Executes images that are external to the current program	—	110
execle	Executes images that are external to the current program	—	110
execv	Executes images that are external to the current program	—	110
execve	Executes images that are external to the current program	—	110
exit, _exit	Terminates the current process	—	112
ftime	Returns the time elapsed since 00:00:00, January 1, 1970 in seconds and milliseconds	timeb	120
getenv	Searches the environment array for the current process and returns the value associated with a specified environment	—	121

Table 6-8: (Cont.) UNIX Emulation Functions

Name	Description	#include Module	Page
getegid	Returns the group and member number from the user identification code	—	123
geteuid	Returns the group and member number from the user identification code	—	123
getgid	Returns the group and member number from the user identification code	—	123
getpid	Returns the current process ID	—	122
getuid	Returns the group and member number from the user identification code	—	123
kill	Sends a signal to a process	—	128
localtime	Converts the time in seconds to hours, minutes, seconds, and so on	—	129
nice	Increases or decreases a process priority	—	131
pause	Suspends the calling process	—	132
sbrk	Adds a number of bytes to the current break address and returns the new break address	—	101
setgid	Included for compatibility; performs no operation	—	147
setuid	Included for compatibility; performs no operation	—	147
time	Returns the time elapsed since 00:00:00, January 1, 1970 in seconds	—	157
times	Returns process times	—	158
umask	Creates a file-protection mask to be used whenever a new file is created	—	160
vfork	Sets up communication channels for spawning and controlling a child process	—	160
wait	Causes the calling process to wait until a signal is received or until one of its child processes terminates	—	162

6.9 Organization of Libraries and Definition (h) Files

All object code for the functions described in this chapter is in the SYS$LIBRARY:CRTLIB.OLB library. The files in this library contain the standard definitions required by the functions.

All text definition files have the file type h. For example, the definition file stdio.h defines the FILE structure used by the standard I/O functions.

Calls to most VAX-11 C library functions are preceded by **#include** control lines, which name a specific definition module. The definition modules required by a function are shown in the synopsis for that function with an **#include** control line. For more details on the **#include** control line, see Chapter 7.

Some VAX-11 C functions are implemented as preprocessor macros for compatibility with other C compilers. To determine whether a function is a true C function or whether it is a macro, see the description of that function.

6.10 Interpreting Synopses of Functions

This chapter follows the usual convention for showing the synopses of functions. A synopsis is a compact representation of the order of a function's parameter list (if any), the parameters' types, and the type of the value returned by the function. The representation closely resembles the format of the actual C text for the function and its parameters.

For example, the synopsis of the **feof** function is:

```
#include stdio
int feof(file_pointer)
FILE *file_pointer;
```

The description of **feof** states that it is implemented as a macro. The synopsis shows that:

- The function requires the definition module stdio.
- The function returns an **int**. Since it is a macro, **feof** is not declared. This line in the synopsis indicates only the return value, not the form of a declaration.
- There is one parameter, file_pointer, which is a pointer to FILE (FILE is defined in stdio).

To use **feof** in a program, you need only write the function call, preceded at some point by the **#include** control line, as in:

```
/* INCLUDE STANDARD DEFINITIONS */
#include stdio
.
.
main()
{
     /* DEFINE A FILE-POINTER */
     FILE *infile;
     .
     .
     /* UNTIL END-OF-FILE REACHED */
     while (!feof(infile))
     /* SOME FILE OPERATION */
        {
        .
        .
        }
}
```

The format of synopses only resembles, and does not duplicate, the format of function definitions. Because some library functions take varying numbers of parameters, synopses have additional conventions not used in actual C function definitions:

- Optional parameters are enclosed in square brackets ([]).
- An ellipsis (...) is used to show that a given parameter may be repeated.
- In cases where the type of a parameter may vary, its type is not shown in the synopsis.

For example:

```
#include stdio
int printf(format_specification[,output_source,...])
char *format_specification;
```

The synopsis for **printf** shows that the parameter output_source is optional, may be repeated, and is not always of the same data type. The remaining information about **printf**'s parameters is in the description of the function.

6.11 Library Functions

The following sections describe each function in the VAX-11 C runtime library. The functions are listed alphabetically.

6.11.1 abort

The function **abort** executes an illegal instruction that terminates the process.

- **Synopsis**

```
abort( )
```

6.11.2 abs, fabs

The function **abs** returns the absolute value of an integer. The function **fabs** returns the absolute value of a floating-point value.

- **Synopses**

```
int abs(integer)
int integer;

#include math
double fabs(x)
double x;
```

6.11.3 access

The function **access** checks a file to see whether a specified access mode is allowed. It returns 0 if the access is allowed and -1 if not. The mode argument is interpreted as shown:

Mode Argument	Access Mode
0	Tests to see if the file exists
1	Execute
2	Write (implies delete access)
4	Read

Combinations of access modes are indicated by summing the above values. For example, 7 indicates RWED.

- **Synopsis**

```
int access(name,mode)
char *name;
int mode;
```

6.11.4 acos

The function **acos** returns a value in the range 0 to pi, which is the arc cosine of its radian argument. The value of acos(x) is 0 when |x|>1, and **errno** is set to EDOM.

- **Synopsis**

```
#include math
double acos(x)
double x;
```

6.11.5 alarm

The function **alarm** sends the signal SIGALRM (defined in the signal module) to the invoking process after the number of seconds indicated by its argument has elapsed. The maximum delay allowed is 2,147,483,647 seconds. Unless it is caught or ignored, the signal terminates the process.

Successive **alarm** calls reinitialize the alarm clock. Alarms are not stacked. Calling **alarm** with a zero argument cancels any pending alarms.

The function returns the number of seconds remaining from a previous alarm request.

Because the clock has a 1-second resolution, the signal may occur up to 1 second early. If the SIGALRM signal is caught, resumption of execution may be delayed by an arbitrary amount because of scheduling delays. See also **pause**, **gsignal**, **ssignal**, **signal**.

- **Synopsis**

```
int alarm(seconds)
unsigned seconds;
```

6.11.6 asin

The function **asin** returns a value in the range -pi/2 to pi/2, which is the arc sine of its radian argument. The value of asin(x) is 0 when |x|>1, and **errno** is set to EDOM.

- **Synopsis**

```
#include math
double asin(x)
double x;
```

6.11.7 atan

The function **atan** returns a value in the range -pi/2 to pi/2, which is the arc tangent of its radian argument.

- **Synopsis**

```
#include math
double atan(x)
double x;
```

6.11.8 atan2

The function **atan2** returns a value in the range -pi to pi. The returned value is the arc tangent of x/y, where x and y are the two arguments.

- **Synopsis**

```
#include math
double atan2(x,y)
double x,y;
```

6.11.9 atof, atoi, atol

These functions convert strings of ASCII characters to the appropriate numeric values. The functions recognize strings in various formats, depending on the returned data type, as follows:

- The string for **atof** may contain leading white space (space, horizontal or vertical tab, carriage return, form feed, or newline). This is followed by an optional sign, then a string of digits (optionally containing a decimal point), then an optional exponent, composed of an 'e' or 'E', and then an (optionally signed) integer:

  [white-spaces][+|-]digits[.digits][e|E[+|-]integer]

  The first unrecognized character ends the string.

- The string for **atoi** and **atol** may contain a series of leading tabs and spaces, then an optional sign, and then a series of digits (with no decimal point):

  [white-spaces][+|-]digits

  atoi and **atol** are synonymous in VAX-11 C.

These functions do not account for overflows resulting from the conversion.

- **Synopses**

```
#include math

double atof(nptr)
char *nptr;

int atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

6.11.10 atoi

See **atof**.

6.11.11 atol

See **atof**.

6.11.12 brk, sbrk

The function **brk** defines the lowest virtual address that is not used by the program. The lowest address is specified by the addr argument, which the function rounds up to the next 512-byte multiple. The rounded address is called the break. This address (the address of a **char**) is returned by the function. An address that is greater than or equal to the break and less than the stack pointer is considered to be outside the program's address space. Attempts to reference it will cause access violations.

sbrk adds the number of bytes specified by its argument to the current break and returns the new break.

When a program is executed, the break is set to the highest location defined by the program and data storage areas. Consequently, **brk** and **sbrk** are needed only by programs that have growing data areas.

brk and **sbrk** return –1 if the program requests too much memory.

- **Synopses**

```
char *brk(addr)
char *sbrk(incr)
unsigned incr, addr;
```

6.11.13 cabs

See **hypot**.

6.11.14 calloc

The function **calloc** allocates an area of memory. The number and size (in bytes) of this area are the arguments. The elements are initialized to 0. If **calloc** is unable to allocate the space, it returns 0.

- **Synopsis**

```
char *calloc(number,size)
unsigned number,size;
```

6.11.15 ceil

The function **ceil** returns the smallest integer that is equal to or greater than its argument.

- **Synopsis**

```
#include math
double ceil(x)
double x;
```

6.11.16 cfree

See **free**.

6.11.17 chdir

The function **chdir** changes the default directory. The function returns 0 if the directory is successfully changed to the given name, and -1 if the change fails. The name argument is a NUL-terminated character string naming a VAX/VMS-style directory.

If **chdir** is called in USER mode, the default directory change is only temporary. On image exit, the default is set to whatever it was before the execution of the image. If you want the change to be effective across images, you should call **chdir** from SUPERVISOR, EXECUTIVE, or KERNEL mode.

- **Synopsis**

```
int chdir(name)
char *name;
```

6.11.18 chmod

The function **chmod** changes the file protection of a file. Only someone with a WRITE privilege for the file can change the mode. The function returns 0 if the change was successful and -1 if unsuccessful.

The first argument is the name of a file. The second argument is a mode. Modes are constructed by ORing any of the following values:

Value	Privilege
0400	OWNER:READ
0200	OWNER:WRITE
0100	OWNER:EXECUTE
0040	GROUP:READ
0020	GROUP:WRITE
0010	GROUP:EXECUTE
0004	WORLD:READ
0002	WORLD:WRITE
0001	WORLD:EXECUTE

When you supply a mode argument of 0, **chmod** gives the file the user's default file protection.

The system is always given the same privileges as the owner. A WRITE privilege also implies a DELETE privilege.

- **Synopsis**

```
int chmod(name,mode)
char *name;
unsigned mode;
```

6.11.19 chown

The function **chown** changes the owner UIC (user identification code) of the file.

The first argument to **chown**, name, is the address of an ASCII file name. The second and third arguments are the owner and group names, respectively. **chown** returns 0 on success and –1 on failure.

- **Synopsis**

```
int chown(name,owner,group)
char *name;
unsigned owner,group;
```

6.11.20 clearerr

The function **clearerr** resets the error and end-of-file indications for a file (so that **ferror** and **feof** will no longer return a nonzero value). **clearerr** is implemented as a macro.

- **Synopsis**

```
#include stdio

clearerr(file_pointer)
FILE *file_pointer;
```

6.11.21 close

The function **close** closes the file associated with a file descriptor. Note that all files are closed on image exit. All buffered data is written to the file if it was opened for writing or update.

The function returns 0 if the file is properly closed. It returns –1 if the file descriptor is undefined or if an error occurs while the file is being closed (for example, if the buffered data cannot be written out).

- **Synopsis**

```
int close(file_descriptor)
int file_descriptor;
```

6.11.22 cos

The function **cos** returns the cosine of its radian argument.

- **Synopsis**

```
#include math
double cos(x)
double x;
```

6.11.23 cosh

The function **cosh** returns the hyperbolic cosine of its argument.

- **Synopsis**

```
#include math
double cosh(x)
double x;
```

6.11.24 creat

The function **creat** creates a new file. The created file has the specification given by the name argument. If the file already exists, a version number one greater than any existing version is assigned to the file.

If the file did not previously exist, it is given the file protection that results from ANDing the mode argument with the complement of the current protection mask (see **umask**). (For details on mode arguments, see **chmod**.) The new file is opened for reading and writing, and its file descriptor is returned. (See also **open**, **close**, **read**, **write**, and **lseek** on file descriptors.)

The function returns an integer file descriptor. It returns -1 to indicate protection violations, undefined directories, and conflicting file attributes.

- **Synopsis**

```
int creat(name,mode[,file_attribute,...])
char *name,*file_attribute;
unsigned mode;
```

name
A NUL-terminated string containing any valid VAX/VMS file specification.

mode
An **unsigned** value that specifies the file-protection mode to be ANDed with the complement of the current protection mode.

file-attribute
A character string of the form:

```
"keyword = value,..."
```

where keyword is an RMS (Record Management Services) field in the file access block (FAB) or record access block (RAB), and value is valid for assignment to that field. Some fields permit you to specify more than one value. In these cases, the values are separated by commas.

The set of valid keywords is listed in Table 6–9.

Table 6-9: File Access Block and Record Access Block Keywords

Keyword	Value	Description
"alq = n"	decimal	Allocation quantity
"bls = n"	decimal	Block size
"deq = n"	decimal	Default extension quantity
"dna = filespec"	string	Default filename string
"fop = val, val,..."		File processing options
	ctg	Contiguous
	cbt	Contiguous-best-try
	tef	Truncate at end-of-file
	cif	Create if nonexistent
	sup	Supersede
	scf	Submit as command file on close
	spl	Spool to system printer on close
	tmd	Temporary delete
	tmp	Temporary (no file directory)
	nef	Not end-of-file
"mbc = n"	decimal	Multiblock count
"mbf = n"	decimal	Multibuffer count
"mrs = n"	decimal	Maximum record size
"rat = val, val..."		Record attributes
	cr	Carriage-return control
	blk	Allow records to span block boundaries
	ftn	FORTRAN print control
	prn	Print file format
"rfm = val"		Record format
	fix	Fixed-length record format
	stm	RMS-11 stream record format
	stmlf	Stream format with line-feed terminator
	stmcr	Stream format with carriage-return terminator
	var	Variable-length record format
	vfc	Variable-length record with fixed control
	udf	Undefined

6.11.25 ctermid

The function **ctermid** returns a character string giving the equivalence string of SYS$COMMAND. This is the name of the controlling terminal. The function takes a single argument, which must be a pointer to **char**. If this argument is null, the file name is stored internally and is overwritten by the next **ctermid** call. Otherwise, the file name is stored beginning at the location pointed to by the argument.

- **Synopsis**

```
#include stdio
char *ctermid(string)
char *string;
```

6.11.26 ctime

ctime converts a time in seconds, since 00:00:00 January 1, 1970, to an ASCII string of the form: wkd mmm dd hh:mm:ss 19yy\n\0.

The argument to **ctime** is a pointer to the time value to be converted. **ctime** returns a pointer to the 26-character ASCII string.

Successive calls to **ctime** overwrite any previous time values.

- **Synopsis**

```
#include time
char *ctime (bintim)
long *bintim
```

6.11.27 cuserid

The function **cuserid** returns a pointer to a character string containing the name of the user who initiated the current process. If the argument is null, the user name is stored internally. If the argument is not null, it points to a storage area of length L_cuserid (defined by stdio), and the name is written into that storage. If the user name is null, the function returns a pointer to a null string.

- **Synopsis**

```
#include stdio
char *cuserid(string)
char *string;
```

- **Example**

Examples 6–1 and 6–2 show two ways to return the user ID with the **cuserid** function.

```
/* WRITE OUT cuserid VALUE */
#include stdio

main()
{
      static char string[L_cuserid] = "";
      printf("Initiating user: %s\n",cuserid(string));
}
```

Example 6–1: Calling cuserid with an Argument

If the user running the program is named ZENO, the program writes the following to **stdout**:

```
Initiating user: ZENO
```

The same output results from the program in Example 6–2.

```
/* WRITE OUT cuserid VALUE */
#include stdio
main()
{
     /* 0 INDICATES NULL ARGUMENT */
     printf("Initiating user: %s\n",cuserid(0));
}
```

Example 6–2: Calling cuserid with the Argument 0

6.11.28 delete

The function **delete** deletes the specified file. The argument is a character string that gives a VAX/VMS file specification. The usual defaults and logical name translation are applied to the file specification.

delete returns 0 if it is successful and -1 if it fails.

- **Synopsis**

```
int delete(file_specification)
char *file_specification;
```

6.11.29 dup, dup2

Given a file descriptor returned by **open**, **creat**, or **pipe**, these functions allocate a new descriptor that refers to the original file. Both return the new file descriptor. **dup2** causes its second argument to refer to the same file as its first argument.

Both functions return -1 if their arguments are invalid. The argument file_descriptor_1 is invalid if it does not describe an open file; file_descriptor_2 is invalid if the new descriptor cannot be allocated.

- **Synopses**

```
int dup(file_descriptor)
int file_descriptor;

int dup2(file_descriptor_1,file_descriptor_2)
int file_descriptor_1,file_descriptor_2;
```

6.11.30 ecvt, fcvt, gcvt

Each of these functions converts its argument, a **double** value, to a NUL-terminated string of ASCII digits and returns the address of the string. In all three functions, value is the double-precision value to be converted, and ndigit is the number of ASCII digits (not including the NUL) to be used in the converted string. Calls to these functions overwrite any existing string.

ecvt and **fcvt** return, via the argument decpt, the position of the decimal point relative to the first character in the returned string. A negative **int** value means that the decimal point is to the left of the returned digits, and a zero means that the decimal point is immediately to the left of the first digit. If **ecvt** and **fcvt** are given a negative value to convert, then the integer pointed to by *sign is set to be nonzero. Otherwise, the integer is set to be zero.

gcvt places the converted string in buf and returns the address of buf. If possible, **gcvt** produces ndigit significant digits in FORTRAN-F format, or if not possible, in E-format. Trailing zeros may be suppressed.

■ **Synopses**

```
char *ecvt(value,ndigit,decpt,sign)
double value;
int ndigit,*decpt,*sign;

char *fcvt(value,ndigit,decpt,sign)
double value;
int ndigit,*decpt,*sign;

char *gcvt(value,ndigit,buf)
double value;
char *buf;
int ndigit;
```

■ **Example**

Example 6–3 shows a program that uses the **ecvt** function to convert a **double** value called val. The program then prints the information returned by **ecvt**.

```
/* ECVT EXAMPLE */
#include stdio
main()

{
     /* VALUE TO BE CONVERTED */
     double val;

     /* VARIABLES FOR SIGN AND DECIMAL PLACE */
     int sign,point;

     /* ARRAY FOR CONVERTED STRING */
     static char string[20];

     val = -3.1297830e-10;

     printf("original value: %e\n",val);
     strcpy(string,ecvt(val,5,&point,&sign));
     printf("converted string: %s\n",string);
     if (sign)
        printf("value is negative\n");
     else printf("value is positive\n");
     printf("decimal point at %d\n",point);
}
```

The output of the program is:

```
original value: -3.129783e-10
converted string: 31298
value is negative
decimal point at -9
```

Example 6–3: The ecvt Function

6.11.31 execl, execv, execle, execve

The functions **execl** and **execv** execute the image in the file named by their first argument. Only VAX-11 C images can be executed by these functions and these images must not be linked with BASE = 0. **execl**'s arguments consist of the file name and character strings giving all arguments for the image. The last argument must be 0, to indicate the end of the list. **execv**'s arguments consist of the name of the file and an array of strings that give the arguments for the image. The last element of the array must be 0. By convention, the first string (arg0 or argv[0]) in both cases is the same as the name of the file.

Any open files remain open across **execv** or **execl** calls. Signals that are ignored in the calling image are also ignored after calls to these functions, but signals for which actions were specified in the calling image revert to their default handling.

There is no return from a successful **execv** or **execl** call, since the calling image is lost. The functions return -1 to indicate a variety of errors, including:

- The file cannot be found.
- The file contents are not executable.
- Nobody (not system, group, owner, or world) has EXECUTE privilege for the file.
- The image requires too much memory.

When a VAX-11 C program executes, it is called as follows by the run-time system:

```
main(argc,argv,envp)
int argc;
char *argv[],*envp[];
{
.
.
}
```

The arguments argc and argv are the argument count and array of argument strings, respectively. The argument envp is an array of strings that specify the program's environment. Each string in envp has the form:

```
name = value
```

where name is either HOME, TERM, PATH, or USER, and value is a NUL-terminated value. The last element of envp must be the null

pointer (0). When the run-time system executes the program, it places a copy of the current environment vector in the external variable environ. This variable is used by **execl** and **execv** to pass the environment to the new program. The meanings of the names are as follows:

- HOME is the user's login directory, that is, the translation of the logical name SYS$LOGIN. The associated value is a VAX/VMS-style directory specification. For example:

```
HOME=DB1:[ZENO]
```

- TERM is the type of terminal being used. The associated values are as follows:

```
la34
la36
la38
la120
vt05
vt52
vt55
vt1xx-80
vt1xx-132
ft1-ft8
unknown
undefined
```

  The values vt1xx-80 and vt1xx-132 indicate any of the VT100-series terminals in 80- or 132-column mode, respectively. The values ft1 through ft8 are for user-designated foreign terminals. A value of unknown indicates that the terminal is not recognized by VAX/VMS; undefined indicates that the terminal is recognized by VAX/VMS but not by VAX–11 C. The null string indicates that there is no terminal associated with the process.

- PATH is the default device and directory. For example, after the DCL command:

```
$ SET DEFAULT WRK$:[ZENO.C.SRC]
```

  the path would be:

```
PATH = WRK$:[ZENO.C.SRC]
```

- USER is the name of the user who initiated the current process, as returned by **cuserid**. For example, if the process is initiated by user ZENO, the user environment string is:

```
USER=ZENO
```

The strings in envp can be retrieved by the function **getenv**, which returns the value associated with the specified name, for example, PATH (see **getenv**).

The functions **execve** and **execle** pass the environment explicitly.

- **Synopses**

```
int execl(name,arg0,arg1,...argn,0)
char *name, *arg0, ... *argn;

int execv(name,argv)
char *name, *argv[ ];

int execle(name,arg0,arg1,...argn,0,envp)
char *name, *arg0, ... *argn,*envp[ ];

int execve(name,argv,envp)
char *name, *argv[ ],*envp[ ];
```

6.11.32 execle

See **execl**.

6.11.33 execv

See **execl**.

6.11.34 execve

See **execl**.

6.11.35 exit, _exit

The functions **exit** and **_exit** terminate the process from which they are called. They return the specified status to the parent process, if any. If the program is invoked by DCL, the status is interpreted by DCL and a message is displayed. The function **exit** flushes and closes all open files before performing the exit; **_exit** terminates the process immediately, without these clean-up actions.

- **Synopsis**

```
exit(status)
int status;

_exit(status)
int status;
```

6.11.36 exp

The function **exp** returns the base e raised to the power of the argument. If an overflow occurs, **exp** returns the largest possible floating-point value and sets **errno** to ERANGE.

- **Synopsis**

```
#include math
double exp(x)
double x;
```

6.11.37 fabs

See **abs**.

6.11.38 fclose

The function **fclose** closes a file by flushing any buffers associated with the file control block and freeing the file control block previously associated with the file pointer.

When a program terminates normally, **fclose** is called automatically for all open files. **fclose** returns 0 on success. If the buffered data cannot be written to the file, or if the file control block is not associated with an open file, **fclose** returns EOF (a preprocessor constant defined in the **#include** module stdio).

- **Synopsis**

```
#include stdio
int fclose(file_pointer)
FILE *file_pointer;
```

6.11.39 fcvt

See **ecvt**.

6.11.40 fdopen

The **fdopen** function associates a file pointer with a file descriptor returned by an **open**, **creat**, **dup**, **dup2**, or **pipe** function. This allows you to access a file originally opened by one of these UNIX I/O functions with standard I/O functions. (Ordinarily, a file can be accessed by either a file descriptor or by a file pointer, but not both, depending on the way it is opened. See Section 6.1.)

The first argument to **fdopen** is the file descriptor returned by **open**, **creat**, **dup**, **dup2**, or **pipe**. The second argument, type, is one of the character strings "r", "w", "a", "r+", "w+", or "a+", for read, write, append, read update, write update, or append update, respectively. (See also **fopen**.) The type must agree with the opened file's access mode (0 = reading, 1 = writing, 2 = reading and writing).

On success, **fdopen** returns a nonzero value which is the file descriptor. On error, **fdopen** returns 0.

- **Synopsis**

```
FILE *fdopen(file_descriptor,type)
int file_descriptor;
char *type;
```

- **Example**

Example 6-4 shows a program that creates a file with variable-length records (rfm = var) and the carriage-return attribute (rat = cr).

The program uses **creat** to create and open the file and **fdopen** to associate the file descriptor with a file pointer. **fdopen** changes the way the file can subsequently be referenced. After the **fdopen** call, the program references the file by file pointer to write records (**fwrite**) and close the file (**fclose**).

```
#include stdio
#define ERROR 0
#define ERROR1 -1
#define BUFFSIZE 132

/* A STREAM FILE USING A FILE DESCRIPTOR
   AND A FILE POINTER */

main()
{
    char buffer[ BUFFSIZE];
    int fildes;
    FILE *fp;

    if ((fildes = creat("data.dat",0,"rat=cr",
                  "rfm=var")) == ERROR1)
        perror("FILE3: create() failed\n");
        exit(2);

    if ((fp = fdopen(fildes,"w")) == NULL)
        perror("FILE3: fdopen() failed\n");
        exit(2);

    while(fgets(buffer,BUFFSIZE,stdin) != NULL)
        {
        if (fwrite(buffer,sizeof(*buffer),
        strlen(buffer),fp) == ERROR)
            perror("FILE3: fwrite() failed\n");
            exit(2);
        }

    if (fclose(fp) == EOF)
        perror("FILE3: fclose() failed\n");
        exit(2);
}
```

Example 6-4: The fdopen Function

6.11.41 feof

The function **feof** tests a file to see if the end-of-file has been reached. If so, **feof** returns a nonzero integer; if not, it returns 0. **feof** is implemented as a macro.

- **Synopsis**

```
#include stdio
int feof(file_pointer)
FILE *file_pointer;
```

6.11.42 ferror

The function **ferror** returns a nonzero integer if an error has occurred while reading or writing a file. A call to the function continues to return this indication until the file is closed or until **clearerr** is called. **ferror** is implemented as a macro.

- **Synopsis**

```
#include stdio
int ferror(file_pointer)
FILE *file_pointer;
```

6.11.43 fflush

The function **fflush** writes out any buffered information for the specified file. (Note that output files are normally buffered if and only if they are not directed to a terminal, but **stderr** is not buffered unless **setbuf** is used.)

fflush returns 0 when it is successful. If the buffered data cannot be written to the file, or if the file control block is not associated with an output file, **fflush** returns EOF (a preprocessor constant defined in the **#include** module stdio).

- **Synopsis**

```
#include stdio
int fflush(file_pointer)
FILE *file_pointer;
```

6.11.44 fgetc

See **getc**.

6.11.45 fgetname

See **getname**.

6.11.46 fgets

See **gets**.

6.11.47 fileno

The function **fileno** returns an integer file descriptor that identifies the specified file. **fileno** is implemented as a macro. See also **open**, **lseek**, **creat**, **read**, and **write**.

- **Synopsis**

```
#include stdio
int fileno(file_pointer)
FILE *file_pointer;
```

6.11.48 floor

The function **floor** returns (as a **double**) the largest integer that is less than or equal to its argument.

- **Synopsis**

```
#include math
double floor(x)
double x;
```

6.11.49 fopen

The function **fopen** opens a file by returning the address of a FILE structure, denoting a file control block. The file control block may be freed with the **fclose** function, or by default on normal program termination.

The first argument to **fopen** is a character string containing a valid VAX/VMS file specification. The second argument, access_mode, is one of the character strings "r", "w", "a", "r+", "w+", or "a+", for read, write, append, read update, write update, or append update, respectively.

The access modes have the following effects:

- "r" opens an existing file for reading.
- "w" opens a file for writing and creates a new file. If the file already exists, it creates a new file with the same name and a higher version number.
- "a" opens the file for append access. An existing file is positioned at end-of-file, and data is written there. If the file does not exist, it is created.

The update access modes allow a file to be opened for both reading and writing. When used with existing files, "r+" and "a+" differ only in the initial positioning within the file. The modes are as follows:

- "r+" opens an existing file for read update access. It is opened for reading, positioned initially at beginning-of-file, but writing is also allowed.
- "w+" opens a new file for write update access.
- "a+" opens a file for append update access. The file is positioned at end-of-file (writing) initially. If the file does not exist, it is created.

The function returns NULL (the null pointer value defined in the **#include** module stdio) to signal errors, including:

- File protection violations
- An attempt to open a nonexistent file for read access
- Failure to open the specified file

■ **Synopsis**

```
#include stdio
FILE *fopen(file_spec,access_mode)
char *file_spec, *access_mode;
```

6.11.50 fprintf

See **printf**.

6.11.51 fputc

See **putc**.

6.11.52 fputs

See **puts**.

6.11.53 fread

The function **fread** reads a specified number of items from the file. The reading begins at the current location in the file. The items read are placed in storage beginning at the location given by the first argument. The size of an item in bytes must also be specified.

The function returns the number of items actually read. If **fread** encounters the end-of-file or an error, it returns 0 (not EOF).

■ **Synopsis**

```
#include stdio
int fread(pointer,size_of_item,number_items,file_pointer)
int number_items,size_of_item;
FILE *file_pointer;
```

The first argument, pointer, points to the items being read. The second argument, size_of_item, is the size of the items being read.

6.11.54 free, cfree

The functions **free** and **cfree** make available for reallocation the area allocated by a previous **calloc**, **malloc**, or **realloc** call. The argument is the address returned by a previous call to **malloc**, **calloc**, or **realloc**. The contents of the area are unchanged. The functions return 0 if the area is successfully freed, -1 if an error occurs. For compatibility with other C implementations, you should use **free** with **malloc** and **cfree** with **calloc**.

- **Synopsis**

```
int free(pointer)
char *pointer;

int cfree(pointer)
char *pointer;
```

6.11.55 freopen

The function **freopen** substitutes the file named by a file specification for the open file addressed by a file pointer. The latter file is closed. The function is typically used to associate one of the predefined names **stdin**, **stdout**, or **stderr** with a file.

The first argument is a pointer to a string that contains a valid VAX/VMS file specification. After the function call, the given file pointer is associated with this file.

The second argument, access_mode, is one of the character strings "r", "w", "a", "r+", "w+", or "a+", for read, write, append, read update, write update, or append update, respectively. (See also **fopen**.)

The third argument is a pointer to a FILE structure, denoting a currently open file. After the function call, the open file is closed.

If the attempt to reopen fails (that is, if the file specified by the first argument cannot be accessed), the function returns the null pointer value. Otherwise, the function returns the address of the reopened file control block.

- **Synopsis**

```
#include stdio
FILE *freopen(file_spec,access_mode,file_pointer)
char *file_spec,*access_mode;
FILE *file_pointer;
```

6.11.56 frexp

The function **frexp** returns the mantissa of a **double** value. The mantissa is a **double** and its magnitude is less than one. The second argument is a pointer to an **int**, to which **frexp** returns the exponent.

- **Synopsis**

```
#include math
double frexp(value,eptr)
double value;
int *eptr;
```

6.11.57 fscanf

See **scanf**.

6.11.58 fseek

The function **fseek** positions the file to the specified byte offset in the file. It returns EOF(a preprocessor constant defined in the **#include** module stdio) for improper seeks, 0 for successful seeks.

- **Synopsis**

```
#include stdio
int fseek(file_pointer,offset,direction)
FILE *file_pointer;
int offset, direction;
```

Direction is an integer indicating whether the offset is measured from the current read or write address (1), from the beginning of the file (0), or from the end-of-file (2).

In general, **fseek** should always be directed to an absolute position returned by **ftell**. With stream files, the direction argument can be 0, 1, or 2. With record files, an **fseek** to a position that was not returned by **ftell** causes unpredictable behavior. See also **ftell**.

6.11.59 ftell

The function **ftell** returns the current byte offset to the specified stream file. The offset is measured from the beginning of the file. With record files, **ftell** returns the position of the next record, not the current byte offset.

This function is useful only for handing an offset to **fseek**, to reposition the file to where it was when **ftell** was called. An error causes –1 to be returned.

- **Synopsis**

```
#include stdio
int ftell(file_pointer)
FILE *file_pointer;
```

6.11.60 ftime

The function **ftime** returns the elapsed time since 00:00:00, January 1, 1970, in a timeb structure. The timeb structure has the members time_t time (which gives the time in seconds), unsigned short millitm (which gives the fractional time in milliseconds), short timezone, and short dstflag. The timezone and dstflag members are always 0.

- **Synopsis**

```
#include timeb
ftime (time_pointer)
struct timeb *time_pointer;
```

6.11.61 fwrite

The function **fwrite** writes a specified number of items to the file. The writing begins at the current location in the file. The size of an item in bytes must also be given.

The function returns the number of items actually written. It returns 0 if there is an error.

- **Synopsis**

```
#include stdio
int fwrite(pointer,size_of_item,number_items,file_pointer)
int number_items, size_of_item;
FILE *file_pointer;
```

The first argument points to the items being written. The second argument is the size in bytes of the items being written.

6.11.62 gcvt

See **ecvt**.

6.11.63 getc, fgetc, getchar, getw

The function **getc** returns the next character as an **int** from a specified file. The file is left positioned after the returned character, and the next **getc** call takes the character from that position. **getc** is implemented as a macro. **fgetc** is identical to **getc**, but it generates an actual function call, not a macro substitution. **getchar** is a macro identical to **getc(stdin)**.

getw returns the next four characters from the specified input file as an **int**. No conversion is performed. If end-of-file is encountered during the retrieval of any of the four characters, then EOF (a preprocessor constant defined in the **#include** module stdio) is returned and all four characters are lost.

All the functions return EOF on end-of-file or error, but since EOF is a perfectly good integer, **feof** and **ferror** should be used to check the success of **getw**.

■ **Synopses**

```
#include stdio

int getc(file_pointer)
FILE *file_pointer;

int fgetc(file_pointer)
FILE *file_pointer;

int getchar()

int getw(file_pointer)
FILE *file_pointer;
```

6.11.64 getchar

See **getc**.

6.11.65 getegid

See **getuid**.

6.11.66 getenv

The function **getenv** searches the environment array for the current process and returns the value associated with a specified environment name.

The names can be one of the following:

- HOME — The user's login directory.
- TERM — The type of terminal being used. See **execl**.
- PATH — The default device and directory.
- USER — The name of the user who initiated the process.

■ **Synopsis**

```
char *getenv(name)
char *name;
```

■ **Example**

```
cfunc()
{
    printf("Terminal type: %s\n",getenv("TERM"));
}
```

If the terminal in use is a DIGITAL VT100 in 132-column mode, the function cfunc writes the following to **stdout**:

```
Terminal type: vt100-132
```

6.11.67 geteuid

See **getuid**.

6.11.68 getgid

See **getuid**.

6.11.69 getname, fgetname

These functions return the VAX/VMS file specification associated with an integer file descriptor (**getname**) or file pointer (**fgetname**).

Both functions place the file specification in a buffer and return the buffer's address. The buffer should be an array large enough to contain a fully qualified file specification (the maximum length is 128 characters). If an error occurs, **getname** returns -1, and **fgetname** returns 0.

- **Synopses**

```
char *getname(file_descriptor,buffer)
int file_descriptor;
char *buffer;

#include stdio
char *fgetname(file_pointer,buffer)
FILE *file_pointer;
char *buffer;
```

6.11.70 getpid

The function **getpid** returns the process ID of the current process.

- **Synopsis**

```
int getpid( )
```

6.11.71 gets, fgets

The function **gets** reads a line from the file **stdin**. The newline character ('\n') that ends the line is replaced by the function with an ASCII NUL character ('\0'). The function returns its argument, which is a pointer to a character string containing the acquired line. If an error occurs or if end-of-file is encountered before a newline is encountered, the function returns the null pointer value.

The function **fgets** reads a line from a specified file, up to a specified maximum number of characters or up to and including the newline character, whichever comes first. The function terminates the line with a NUL ('\0') character. Note that, unlike **gets**, **fgets** places the newline that terminates the input record into the user buffer if it fits. On end-of-file or error, the function returns the NULL pointer value (defined in the **#include** module stdio). Otherwise, it returns the address of the first character in the line.

- **Synopses**

```
#include stdio
char *gets(string)
char *string;

char *fgets(string,maxline,file_pointer)
char *string;
int maxline;
FILE *file_pointer;
```

6.11.72 getuid, getgid, geteuid, getegid

These functions return, in VAX/VMS terms, group and member numbers from the user identification code (UIC). (For example, if the UIC is [313,031], 313 is the group number, and 031 is the member number.)

In VAX-11 C, there is no difference between **getgid** and **getegid**. Both return the group number from the current UIC. Similarly, **getuid** and **geteuid** both return the member number from the current UIC.

- **Synopses**

```
unsigned getgid()
unsigned getegid()
unsigned getuid()
unsigned geteuid()
```

6.11.73 getw

See **getc**.

6.11.74 gsignal

The function **gsignal** raises (generates) a specified software signal. Raising a signal causes the action established by the **ssignal** function to be taken.

The argument to **gsignal**, sig, identifies the signal to be raised. The result of a **gsignal** call is one of the following:

- If **gsignal** specifies a sig argument that is outside the range defined in the signal module, then **gsignal** returns 0, and **errno** is set to EINVAL.
- If **ssignal** establishes SIG_DFL (default action) for the signal, then **gsignal** does not return. The image is exited with the VAX/VMS error code that corresponds to the signal.
- If **ssignal** establishes SIG_IGN (ignore signal) as the action for the signal, then **gsignal** returns its argument, sig.
- Otherwise, **ssignal** must have established an action function for the signal. That function is called, and that function's return value is returned by **gsignal**.

■ **Synopsis**

```
#include signal
int gsignal(sig)
int sig;
```

6.11.75 hypot, cabs

The functions **hypot** and **cabs** return:

```
sqrt(x*x + y*y)
```

■ **Synopsis**

```
#include math

double hypot(x,y)
double x,y;

double cabs(z)
struct
    {
    double x,y;
    } z;
```

6.11.76 isalnum

The macro **isalnum** returns a nonzero integer if its argument is one of the alphanumeric ASCII characters.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isalnum(character)
char character;
```

6.11.77 isalpha

The macro **isalpha** returns a nonzero integer if its argument is an alphabetic ASCII character.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isalpha(character)
char character;
```

6.11.78 isascii

The macro **isascii** returns a nonzero integer if its argument is any ASCII character.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isascii(character)
char character;
```

6.11.79 isatty

The function **isatty** returns 1 if the specified file descriptor is associated with a terminal, and 0 if it is not. A return value of -1 indicates an error, for example, the file descriptor is not associated with an open file.

- **Synopsis**

```
int isatty(file_descriptor)
int file_descriptor;
```

1. Refer to Appendix G.

6.11.80 iscntrl

The macro **iscntrl** returns a nonzero integer if its argument is an ASCII DEL character (177 octal) or any nonprinting ASCII character (code less than 40 octal).[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int iscntrl(character)
char character;
```

6.11.81 isdigit

The macro **isdigit** returns a nonzero integer if its argument is a decimal digit character (0–9).[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isdigit(character)
char character;
```

6.11.82 isgraph

The macro **isgraph** returns a nonzero integer if its argument is a graphic ASCII character.[1] Otherwise, it returns 0. Graphic ASCII characters are those with octal codes greater than or equal to 41 ('!') and less than or equal to 176 ('~'). In other words, they comprise the set of printable characters minus the space.

- **Synopsis**

```
#include ctype
int isgraph(character)
char character;
```

6.11.83 islower

The macro **islower** returns a nonzero integer if its argument is a lowercase alphabetic ASCII character.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int islower(character)
char character;
```

1. Refer to Appendix G.

6.11.84 isprint

The macro **isprint** returns a nonzero integer if its argument is any ASCII printing character (ASCII codes from 40 octal to 176 octal).[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isprint(character)
char character;
```

6.11.85 ispunct

The macro **ispunct** returns a nonzero integer if its argument is an ASCII punctuation character — that is, if it is nonalphanumeric and greater than 40 octal.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int ispunct(character)
char character;
```

6.11.86 isspace

The macro **isspace** returns a nonzero integer if its argument is "whitespace", that is, it is an ASCII space, tab (horizontal or vertical), carriage-return, form-feed, or newline character.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isspace(character)
char character;
```

6.11.87 isupper

The macro **isupper** returns a nonzero integer if its argument is an uppercase alphabetic ASCII character.[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isupper(character)
char character;
```

1. Refer to Appendix G.

6.11.88 isxdigit

The macro **isxdigit** returns a nonzero integer if its argument is a hexadecimal digit (0–9, A–F, or a–f).[1] Otherwise, it returns 0.

- **Synopsis**

```
#include ctype
int isxdigit(character)
char character;
```

6.11.89 kill

The function **kill** sends a signal to the process specified by a process ID. Unless the user has system privileges, the sending and receiving processes must have the same UIC. The function returns 0 if the **kill** was successfully queued. It returns –1 to indicate errors, including:

- The receiving process has a different UIC and the user is not a SYSTEM user.
- The receiving process does not exist.

See also **ssignal, gsignal, getpid.**

- **Synopsis**

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

6.11.90 ldexp

The function **ldexp** returns its first argument multiplied by 2 raised to the power of its second argument, that is, $x(2^e)$.

If underflow occurs, **ldexp** returns 0, and if overflow occurs, it returns the largest possible value of the appropriate sign. In both cases, **errno** is set to ERANGE.

- **Synopsis**

```
#include math

double ldexp(x,e)
double x;
int e;
```

1. Refer to Appendix G.

6.11.91 localtime

The **localtime** function converts a time (expressed as the number of seconds elapsed since 00:00:00 January 1, 1970) into hours, minutes, seconds, and so on. The converted time value is placed in a time structure defined in the time **#include** module with the tag tm. The following member names are offsets into the structure. They are integers:

- tm_hour — hours (24)
- tm_min — minutes
- tm_sec — time in seconds
- tm_isdst — daylight savings time (always 0)
- tm_mon — month (0–11)
- tm_mday — day of the month (1–31)
- tm_year — year (last two digits)
- tm_wday — day of the week (0–6)
- tm_yday — day of the year (0–365)

The argument to **localtime** is a pointer to the time in seconds relative to 00:00:00 January 1, 1970. This time can be generated by the **time** function or supplied by the user. **localtime** returns a pointer to the time structure. Successive calls to **localtime** overwrite the structure.

■ **Synopsis**

```
#include time
struct tm *localtime(bintim)
int *bintim;
```

6.11.92 log, log10

The function **log** returns the natural (base e) logarithm of the argument, which must be of type **double**. (The returned value is also **double**.) **log10** returns the **double** base 10 logarithm of its **double** argument. If the argument x is zero or negative, the functions return 0 and set **errno** to EDOM.

■ **Synopses**

```
#include math

double log(x)
double x;

double log10(x)
double x;
```

6.11.93 longjmp

See **setjmp**.

6.11.94 lseek

The function **lseek** positions a file to an arbitrary byte position and returns the new position as an **int**. The function sets the new position relative either to the beginning of the file (direction=0), the current position (direction=1), or the end of the file (direction=2). The file descriptor is an integer returned by **open**, **creat**, **dup**, or **dup2**. The offset argument and the return value are measured in bytes. See also **open**, **creat**, **dup**, **dup2**, and **fseek**.

The function returns -1 if the file descriptor is undefined or if you attempt to seek before the beginning of the file.

lseek can position a stream file on any byte offset. **lseek** can position a record file only on record boundaries. The available standard I/O functions always position a record file at its first byte, at the end-of-file, or, in the case of **fwrite** and **fread**, on the record boundary following the last record that was written or read. Therefore, the arguments given to **lseek** must specify either the beginning or end of the file, a zero offset from the current position (an arbitrary record boundary), or the position returned by a previous, valid **lseek** call.

CAUTION

> If you seek beyond the end-of-file, and then write to the file, the function creates a "hole" by filling the skipped bytes with zeros.
>
> In general, **lseek** should always be directed to absolute positions returned by **ftell**. With stream files, the direction argument can be 0, 1, or 2. With record files, an **lseek** to a position that was not returned by **ftell** causes unpredictable behavior.

- **Synopsis**

```
int lseek(file_descriptor,offset,direction)
int offset,file_descriptor,direction;
```

The following call obtains the position of the next record in an RMS record file (which has the descriptor file1):

```
/* 0 RELATIVE TO CURRENT POSITION */
pos = lseek(file1,0,1)
```

The return value in pos can then be used later in the program (perhaps after the file has been repositioned by **write** or **read**) to return to this position, as in:

```
/* POSITION RELATIVE TO BEGINNING */
newpos = lseek(file1,pos,0);
```

6.11.95 malloc

The function **malloc** allocates a contiguous area of memory whose size in bytes is supplied as an argument. It returns the address of the first byte, which is aligned on a longword boundary. **malloc** returns 0 if it is unable to allocate enough memory.

- **Synopsis**

```
char *malloc(size)
unsigned size;
```

6.11.96 mktemp

The **mktemp** function creates a unique file name from a template. You supply the template in the form, "namXXXXXX". The six trailing X's are replaced by a unique series of characters. You may supply the first three characters (nam).

The argument to **mktemp** is a pointer to the template. **mktemp** returns a pointer to the file name it creates. If a unique file name cannot be created, **mktemp** returns a pointer to an empty string (\0).

- **Synopsis**

```
char *mktemp(template)
char *template;
```

6.11.97 modf

The function **modf** accepts two arguments, a **double** value and a pointer to an **int**. It returns the positive fractional part of its first argument and assigns the address of the integral part to its second argument.

- **Synopsis**

```
#include math

double modf(value,iptr)
double value,*iptr;
```

6.11.98 nice

The function **nice** increases or decreases process priority by the amount of the argument. A positive argument decreases priority, and a negative argument increases priority. The resulting priority cannot be less than one or greater than the process's base priority. **nice** returns 0 on success and –1 on failure.

When a process forks, the resulting child inherits the parent's priority.

- **Synopsis**

```
nice(increment)
int increment;
```

6.11.99 open

The function **open** opens a file by file specification, either for reading (mode = 0), writing (mode = 1), or update (both reading and writing, mode = 2). It returns an integer file descriptor that is used by **read**, **write**, **lseek**, **dup**, **dup2**, and **close**.

The function positions the file at its beginning (byte 0). It returns -1 if the file does not exist, if it is protected against reading or writing, or if the file, for any other reason, cannot be opened.

See also **creat**, **read**, **write**, **close**, **dup**, **dup2**, and **lseek**.

NOTE

If you intend to do random writing to a stream file, the file *must* be opened for update (mode = 2).

- **Synopsis**

```
int open(name,mode)
char *name;
int mode;
```

The argument, name, is a NUL-terminated character string containing a valid VAX/VMS file specification.

6.11.100 pause

The function **pause** causes its calling process to stop (hibernate) until the process receives a signal. Control is not returned to the process that called **pause**, except after a SYS$WAKE system service call. The process may be reawakened by **kill** or **alarm**.

See also **signal**.

- **Synopsis**

```
pause( )
```

6.11.101 perror

The function **perror** writes a short error message to **stderr** describing the last error encountered during a call to the C run-time library from a C program. It writes out its argument (a user-supplied prefix to the error message), followed by a colon, followed by the message itself, followed by a newline. The argument typically is the name of the program that incurred the error.

The message written by **perror** is taken from the standard message string vector **sys_errlist**; **sys_errlist** can be indexed by the value of the external variable **errno**.

The variable **sys_nerr** contains the current number of messages in **sys_errlist**.

- **Synopsis**

```
extern char *sys_errlist[];
extern int sys_nerr;

perror(string)
char *string;
```

6.11.102 pipe

The **pipe** function allows two processes, which are spawned by subsequent **vfork** calls, to exchange data with **read** and **write** calls. It returns 0 if the pipe was created and -1 if the attempt to create the pipe failed.

After a successful return, the array file_descriptor contains two file descriptors. The first descriptor (in element 0) is used for reading data from the pipe, and the second (in element 1) is used for writing data to the pipe. The maximum size of a single write is 512 bytes.

The forked processes inherit the open file descriptors.

- **Synopsis**

```
int pipe(file_descriptor)
int file_descriptor[2];
```

6.11.103 pow

The function **pow** returns the first argument raised to the power of the second argument. Both arguments must be **double**, and the returned value is **double**. If the result overflows, **pow** returns the largest possible floating-point value and sets **errno** to ERANGE. If the argument y is negative and nonintegral, or if both arguments are zero, **pow** returns 0 and sets **errno** to EDOM.

- **Synopsis**

```
#include math

double pow(x,y)
double x,y;
```

6.11.104 printf, fprintf, sprintf

These functions perform formatted output to the standard output (**printf**), to a specified file (**fprintf**), or to a character string in memory (**sprintf**). All three take a format-specification argument that contains characters to be written literally to the output and/or conversion specifications that correspond to the list of optional output sources.

All three functions return the number of characters actually written out. **printf** and **fprintf** return -1 if an I/O error occurs. **sprintf** returns -1 if the output string overflowed the internal buffer.

- **Synopses**

```
#include stdio

int printf(format_specification[,output_source,...])
char *format_specification;

int fprintf(file_pointer,format_specification[,output_source,...])
FILE *file_pointer;
char *format_specification;

int sprintf(string,format_specification[,output_source,...])
char *string, *format_specification;
```

In the **printf**, **fprintf**, and **sprintf** functions, the output sources are expressions whose types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the output sources may be omitted. Otherwise, the function call must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources. Conversion specifications are matched to output sources in simple left-to-right order.

The format specification is a character string that specifies the output format. The string may contain two kinds of items:

- Ordinary characters, which are simply copied to the output.
- Conversion specifications, each of which causes the conversion of a corresponding output source to a character string, in a particular format.

■ Conversion Specifications

Each conversion specification begins with a percent sign (%) and ends with a conversion character that specifies an output format. The output formats are are described in Table 6–10.

Table 6–10: Conversion Characters for Formatted Output

Character	Meaning
d	Convert to decimal format.
o	Convert to unsigned octal format (without leading zero).
x	Convert to unsigned hexadecimal format (without leading 0x). An uppercase X causes the hexadecimal digits A–F to be printed in uppercase. A lowercase x causes those digits to be printed in lowercase.
u	Convert to unsigned decimal format (giving a number in the range 0 to 4,294,967,295).
c	Output single character (NUL characters are ignored).
s	Output as character string (write out characters until NUL is encountered or until number of characters indicated by the precision specification is exhausted. If the precision specification is zero or omitted, all characters up to a NUL are output).
e	Convert **float** or **double** to the format [–]m.nnnnnnE[+ | –]xx, where the number of n's is specified by the precision (default = 6). If the precision is explicitly zero, the decimal point appears but no n's appear. An E is printed if the conversion character is an uppercase E. An e is printed if the conversion character is a lowercase e.
f	Convert **float** or **double** to the format [–]m..m.nnnnnn, where the number of n's is specified by the precision (default = 6). Note that the precision does not determine the number of significant digits printed. If the precision is explicitly zero, no decimal point appears and no n's appear.
g	Convert **float** or **double** to d, e, or f format, whichever is shorter (suppress insignificant zeros).
%	Write out the % symbol. No conversion is performed.

The following characters can be used between the % sign and the conversion character. They are optional, but if specified, they must occur in the order listed.

Character	Meaning
-	Left-justify the converted output source in its field.
width	Use this integer constant as the minimum field width. If the converted output source is wider than this minimum, write it out anyway. If the converted output source is narrower than the minimum width, pad it to make up the field width. Padding is with spaces normally, and with zeros if the field width is specified with a leading zero. (This does not mean that the width is an octal number.) Padding is on the left normally and on the right if a minus sign is used.
.	Separates field width from precision.
precision	Use this integer constant to designate the maximum number of characters to print with s format, or the number of fractional digits with e or f format.
l	Indicates that a following d, o, x, or u specification corresponds to a **long** output source. (Note that, in VAX–11 C, all **int**s are long by default.)
*	Can be used to replace the field width specification and/or the precision specification. The corresponding width or precision is given in the output source.

■ Example

Example 6–5 shows how **printf** interprets different kinds of conversion specifications.

```
#include stdio

main()

{
    double val   = 123.3456e+3;
    char   c     = 'C';
    int    i     = -1500000000;
    char  *s     = "thomasina";

/* Print the format code, a colon, two tabs,
*  and the formatted output value, with the
*  output field delimited by <>.
*/

    printf("%%9.4f:\t\t<%9.4f>\n",val);
    printf("%%9f:\t\t<%9f>\n",val);
    printf("%%9.0f:\t\t<%9.0f>\n",val);
    printf("%%-9.0f:\t\t<%-9.0f>\n\n",val);

    printf("%%11.6e:\t\t<%11.6e>\n",val);
    printf("%%11e:\t\t<%11e>\n",val);
    printf("%%11.0e:\t\t<%11.0e>\n",val);
    printf("%%-11.0e:\t\t<%-11.0e>\n\n",val);

    printf("%%11g:\t\t<%11g>\n",val);
    printf("%%9g:\t\t<%9g>\n\n",val);

    printf("%%d:\t\t<%d>\n",c);
    printf("%%c:\t\t<%c>\n",c);
    printf("%%o:\t\t<%o>\n",c);
    printf("%%x:\t\t<%x>\n\n",c);

    printf("%%d:\t\t<%d>\n",i);
    printf("%%u:\t\t<%u>\n",i);
    printf("%%x:\t\t<%x>\n\n",i);

    printf("%%s:\t\t<%s>\n",s);
    printf("%%-9.6s:\t\t<%-9.6s>\n",s);
    printf("%%-*.*s:\t\t<%-*.*s>\n",9,5,s);
    printf("%%6.0s:\t\t<%6.0s>\n\n",s);

}
```

Example 6-5: The printf Function

The program writes the following to **stdout**:

```
%9.4f:      <123345.6000>
%9f:        <123345.600000>
%9.0f:      <   123346>
%-9.0f:     <123346   >

%11.6e:     <1.233456e+05>
%11e:       <1.233456e+05>
%11.0e:     <      1.e+05>
%-11.0e:    <1.e+05      >

%11g:       <      123346>
%9g:        <   123346>

%d:         <67>
%c:         <C>
%o:         <103>
%x:         <43>

%d:         <-1500000000>
%u:         <2794967296>
%x:         <a697d100>

%s:         <thomasina>
%-9.6s:     <thomas   >
%-*.*s:     <thoma    >
%6.0s:      <thomasina>
```

Example 6-5: (Cont.) The printf Function

6.11.105 putc, fputc, putchar, putw

The function **putc** writes a single character to a file and returns the character. The file is left positioned after the character. **putc** is implemented as a macro; **fputc** acts the same as **putc** but is a true function, not a macro. The function **putchar** is implemented as a macro. It writes a single character to the standard output (**stdout**) and returns the character. **putchar** is identical to **putc(stdout)**.

putw writes four characters to the output file as an **int**. No conversion is performed. If end-of-file is encountered during the retrieval of any of the four characters, then EOF is returned and all four characters are lost.

All of these functions return EOF (defined in the **#include** module stdio) to designate output errors. Since EOF is itself an integer, **ferror** should be used to detect errors encountered by **putw**.

- **Synopses**

```
#include stdio

int putc(character,file_pointer)
char character;
FILE *file_pointer;

int fputc(character,file_pointer)
char character;
FILE *file_pointer;

int putchar(character)
char character;

int putw(integer,file_pointer)
int integer;
FILE *file_pointer;
```

6.11.106 putchar

See **putc**.

6.11.107 puts, fputs

The function **puts** writes a character string to **stdout**, followed by a newline. The function **fputs** writes a character string to a specified file. It does not append a newline to the string. Neither function copies the terminating NUL to the output stream.

- **Synopses**

```
#include stdio

int puts(string)
char *string;

int fputs(string,file_pointer)
char *string;
FILE *file_pointer;
```

6.11.108 putw

See **putc**.

6.11.109 rand, srand

The function **rand** returns pseudorandom numbers in the range 0 to 2^{31}–1. It uses a multiplicative congruential random number generator with a repeat factor (period) of 2^{32}. The random number generator is reinitialized by calling **srand** with the argument 1, or it can be set to a specific point by calling **srand** with any other number.

- **Synopses**

```
int rand( );

int srand(seed)
int seed;
```

6.11.110 read

The function **read** reads bytes from a file and places them in a buffer. The buffer argument is the address of at least n bytes of contiguous storage in which the input is placed. The function returns the number of bytes actually read. The return value does not necessarily equal nbytes. For example, if the input is from a terminal, at most one line of characters is read.

A return value of 0 means that end-of-file was encountered. A return value of –1 indicates any sort of read error, including physical input errors, illegal buffer addresses, protection violations, undefined file descriptors, and so forth.

The specified file descriptor must refer to a file currently opened for reading (see **open**).

- **Synopsis**

```
int read(file_descriptor,buffer,nbytes)
int file_descriptor,nbytes;
char *buffer;
```

6.11.111 realloc

The function **realloc** changes the size of the area pointed to by the first argument to the number of bytes given by the second argument. **realloc** returns the address of the area, since the area may have moved to a new address. If the area was moved, the space previously occupied is freed. If **realloc** is unable to reallocate the space (for example, if there is not enough room), it returns 0.

The contents of the area are unchanged up to the lesser of the old and new sizes. New space in the reallocated area is initialized with 0.

The first argument may point to an allocated area or, unless other allocations have been made in the meantime, to an area freed by **free** or **cfree**.

- **Synopsis**

```
char *realloc(pointer,size)
char *pointer;
unsigned size;
```

6.11.112 rewind

The function **rewind** sets the file to the beginning. **rewind** is equivalent to **fseek**(file-pointer,0,0). It returns -1 to indicate failure; 0 to indicate success. **rewind** can be used with either record or stream files.

- **Synopsis**

```
#include stdio
int rewind(file_pointer)
FILE *file_pointer;
```

6.11.113 sbrk

See **brk**.

6.11.114 scanf, fscanf, sscanf

These functions perform formatted input from the standard input (**scanf**), from a specified file (**fscanf**), or from a character string in memory (**sscanf**). All three take a format-specification argument that contains ordinary characters, which must match the corresponding characters in the input, and/or conversion specifications that correspond to the list of optional input targets.

The functions return the number of successfully matched and assigned input items. If end-of-file (or string) is encountered, the functions return EOF (a preprocessor constant defined in the **#include** module stdio).

■ **Synopses**

```
#include stdio

int scanf(format_specification[,input_pointer,...])
char *format_specification;

fscanf(file_pointer,format_specification[,input_pointer,...])
FILE *file_pointer;
char *format_specification;

sscanf(string,format_specification[,input_pointer,...])
char *string,*format_specification;
```

In all three functions, the format specification is a character string specifying the input formats to be used. A format specification can include three kinds of items:

1. White-space characters (spaces, tabs, and newlines), which match optional white-space characters in the input field.
2. Ordinary characters (not %), which must match the next non-white-space character in the input.
3. Conversion specifications, which govern the conversion of the characters in an input field and their assignment to an object indicated by a corresponding input pointer (see list below).

Each input pointer is an address expression indicating an object whose type matches that of a corresponding conversion specification. (Conversion specifications form part of the format specification.) The indicated object is the target that receives the input value. There must be as many input pointers as there are conversion specifications, and the addressed objects must match the types of the conversion specifications.

■ **Conversion Specifications**

Each conversion specification begins with a percent sign (%). This sign is followed by an optional assignment-suppression character (*) — see "Remarks" — an optional number giving the maximum field width, and a conversion character. The conversion characters are described in Table 6–11.

■ **Remarks**

- The delimiters of the input field can be changed with the bracket ([ ]) conversion specification, described above. Otherwise, an input field is defined as a string of non-white-space characters. It extends either to the next white-space character or until the field width, if specified, is exhausted. (Note, therefore, that the function reads across line/record boundaries, since the newline character is a white-space character.)
- A call to one of these functions resumes searching immediately after the last character processed by a previous call.

Table 6–11: Conversion Characters for Formatted Input

Character	Meaning
d	Expect a decimal integer in the input. The corresponding argument must point to an **int**.
o	Expect an octal integer in the input (with or without a leading zero). The corresponding argument must point to an **int**.
x	Expect a hexadecimal integer in the input (without a leading 0x). The corresponding argument must point to an **int**.
c	Expect a single character in the input. The corresponding argument must point to a **char**. The usual skipping of white space is disabled in this case, so that *n* white-space characters can be read with *%nc*. If a field width is given with c, the given number of characters is read, and the corresponding argument should point to an array of **char**.
s	Expect a character string in the input. The corresponding argument must point to an array of characters that is large enough to contain the string plus the terminating NUL character (\0). The input field is terminated by a space, tab, or newline.
f	Expect a floating-point number in the input. The corresponding argument must point to a **float**. The input format for floating-point numbers is [+ ¦ -]nnn[.[ddd]][{E ¦ e}[+ ¦ -]nn], where the n's and d's are decimal digits (as many as indicated by the field width minus the signs and the letter E).
e	Synonym for f.
ld,lo,lx	Same as d, o, x, except that a **long** integer of the specified radix is expected. (Retained for compatibility only, since **long** and **int** are the same in VAX-11 C.) The same effect can be achieved by using D, O, and X.
le,lf	Same as e, f, except that the corresponding argument is a **double** instead of a **float**. (The same effect can be achieved by using an uppercase E or F.)
hd,ho,hx	Same as d,o,x, except that a **short** integer of the specified radix is expected.
[..]	Expect a string that is not delimited by white-space characters. The brackets enclose a set of characters (not a string). Ordinarily, this set (or "character class") is made up of the characters that comprise the string field. Any character not in the set will terminate the field. However, if the first (leftmost) character is an up-arrow (^), then the set shows the characters that terminate the field. The corresponding argument must point to an array of characters.

- If the assignment-suppression character (*) appears in the format specification, no assignment is made. The corresponding input field is interpreted and then skipped.
- The arguments must be pointers or other address-valued expressions, since C permits only calls by value. To read a number in decimal format and assign its value to n, you must write

```
scanf("%d",&n)
```

not

```
scanf("%d",n)
```

- White space in a format specification matches optional white space in the input field. That is, the format specification:

```
field = %x
```

matches:

```
field = 5218
field=5218
field= 5218
field =5218
```

but not:

```
fiel d=5218
```

6.11.115 setbuf

The function **setbuf** associates a buffer with an input or output file. It may be used after the file has been opened, but must be used before any I/O is done to it. It causes file operations to use the specified character array as a buffer instead of using an automatically allocated buffer. The buffer must be large enough to hold an entire input record. The BUFSIZ constant defined in the stdio module is available for you to use as the size of the buffer. If the buffer is NULL (defined in the **#include** module stdio), the file will be unbuffered. Otherwise the buffer is used for all subsequent I/O operations on the file. A buffer is normally obtained by calling **malloc**.

■ **Synopsis**

```
#include stdio
setbuf(file_pointer,buffer)
FILE *file_pointer;
char *buffer;
```

6.11.116 setgid

See **setuid**.

6.11.117 setjmp, longjmp

The **setjmp** and **longjmp** function pair provides a way to transfer control from a nested series of functions back to a predefined point without returning normally (that is, not by a series of **return** statements). The **setjmp** function saves the context of the calling function in an environment buffer. The **longjmp** function restores the context of the environment buffer.

The environment buffer is declared as an array of integers long enough to hold the register context of the calling function. It is declared by the **typedef** jmp_buf. The contents of the general-purpose registers, including the program counter (PC), are stored in the buffer.

When **setjmp** is first called, it returns the value 0. If **longjmp** is then called, naming the same environment as the call to **setjmp**, control is returned to the **setjmp** call as if it had returned normally a second time. The return value of **setjmp** in this second return is the value supplied by the user in the **longjmp** call. To preserve the true value of **setjmp**, the function calling **setjmp** must not be called again until the associated **longjmp** is called.

- **Synopses**

```
#include setjmp

setjmp(env)
jmp_buf env;

longjmp(env,val)
jmp_buf env;
```

- **Example**

The program in Example 6–6 uses a series of **case** statements to determine how the program has returned from a series of functions. The NORMAL case is always executed once. The program sets the return value of **setjmp** to 0 and calls the function x, which calls y. If the function y succeeds, it calls z. If it does not succeed, y uses the **longjmp** function to return the value Y_FAILED to the main function. If the function z succeeds, it executes a series of **return** statements to return to the middle of the NORMAL case. Otherwise, z uses the **longjmp** function to return the value Z_FAILED.

```
#include stdio
#include setjmp
#define NORMAL 0
#define Y_FAILED 1
#define Z_FAILED 2
jmp_buf environment;

FILE *fp;

main()
{
     switch (setjmp(environment))
     {
     case NORMAL:
          fp = fopen("anyfile","w");
          x();
          fclose(fp);
          break;

     case Y_FAILED;
          fprintf (fp,"Could not proceed; failed in y().\n");
          y_cleanup();
          break;

     case Z_FAILED;
          fprintf(fp,"Could not proceed; failed in z().\n");
          fclose(fp);
          break;
     }
}
x()
{
     .
     .
     y();
     .
}
y()
{
     .
     .
     if("error")
          longjmp(environment, Y_FAILED);
          /* CONTROL GOES BACK TO THE BEGINNING
             OF THE case STATEMENT */
     z();
}
```

```
z()
{
     .
     .
     if("error")
          longjmp(environment, Z_FAILED);
          /* CONTROL GOES BACK TO THE BEGINNING
             OF THE case STATEMENT */
}
```

Example 6-6: The setjmp and longjmp Functions

6.11.118 setuid, setgid

These functions are included only for program compatibility. They both return 0 (to indicate success). They perform no other operation.

- **Synopses**

```
int setuid(member_number)
unsigned member_number;

int setgid(group_number)
unsigned group_number;
```

6.11.119 signal

The function **signal** allows you either to catch or to ignore a signal. Signals are raised by a variety of events, including:

- A **gsignal** call (see **gsignal**).
- A user typing CTRL/C at a terminal (thus raising the signal SIGINT).
- Certain programming errors.
- A **kill** function call from another process.

Signals are given the mnemonics (as in SIGINT, above) found in the definition module signal. Normally, all signals cause the termination of the receiving process. However, the **signal** function allows you to ignore most of them or to interrupt to a specific location for handling. Table 6-12 shows the signals defined in the signal module, ways to generate the signals on VAX/VMS, and the qualities of the signal, such as whether or not the signal can be ignored. (Unless noted otherwise, the signal can be reset and it can be caught or ignored.)

Table 6–12: VAX-11 C Signals

Name	Description	Generated by	Notes
SIGHUP	Hang up	Data set hang up	
SIGINT	Interrupt	VMS CTRL/C interrupt	
SIGQUIT	Quit	CTRL/C if the action for SIGINT is the SIG_DFL default	
SIGILL	Illegal instruction	Illegal instruction, reserved operand, or reserved address mode	Not reset when caught
SIGTRAP	Trace trap	TBIT trace trap or breakpoint fault instruction	Not reset when caught
SIGIOT	IOT instruction	Not implemented	
SIGEMT	EMT instruction	Compatibility mode trap or op code reserved to customer	
SIGFPE	Floating-point exception	Floating-point overflow/underflow	
SIGKILL	Kill	External signal only	Cannot be caught or ignored
SIGBUS	Bus error	Access violation or change mode user	
SIGSEGV	Segment violation	Length violation or change mode supervisor	
SIGSYS	System call error	Bad argument to system call	
SIGPIPE	Broken pipe	Not implemented	
SIGALRM	Alarm clock	Timer AST	
SIGTERM	Software terminate	External signal only	

The first argument in a **signal** call is sig, the number or mnemonic associated with a signal. Customarily, the sig argument is one of the mnemonics defined in the signal module. The second argument is func, which is either the action to be taken when the signal is raised, or the address of a function needed to handle the signal.

If the func argument is the constant SIG_DFL (defined in the **#include** module signal), the action for the given signal is reset to the default action, which is the termination of the receiving process. If the argument is SIG_IGN, the signal is ignored. (Note that not all signals can be ignored.)

If the func argument is neither SIG_DFL nor SIG_IGN, then it specifies the address of a signal-handling function. When the signal is raised, the addressed function is called with sig as its argument. When the addressed function returns, the interrupted process continues at the point of interruption. (This is called "catching a signal.") Except as indicated in Table 6–12, signals are reset to SIG_DFL after they have been caught. Thus, you must call **signal** each time you want to catch a signal.

The **signal** function returns the address of the function previously (or initially) established to handle the signal. If the sig argument is out of range, **signal** returns –1, and **errno** is set to EINVAL.

A child process (see **vfork**) inherits only the defaulted and ignored signals of the process that spawned it.

An **exec** call (see **exec**) resets all caught signals to the default action.

- **Synopsis**

```
#include signal

int (*signal(sig,func))( )          /* FUNCTION RETURNING
                                       ADDRESS OF FUNCTION
                                       RETURNING int */
int sig;                            /* SIGNAL NUMBER */
int (*func)( );                     /* ADDRESS OF FUNCTION
                                       RETURNING int */
```

- **Example**

Example 6–7 shows how you can use the **signal**, **alarm**, and **pause** functions to alternately suspend and resume a program.

```
#define SECONDS 5

#include stdio
#include signal

/* INITIALIZE THE ALARM COUNTER */
int number_of_alarms = 5;

main()
{
     int alarm_action();

/* PASS SIGNAL AND FUNCTION TO signal */
     signal(SIGALRM,alarm_action);

/* SET ALARM CLOCK FOR 5 SECONDS */
     alarm(SECONDS);

/* SUSPEND THE PROCESS UNTIL THE SIGNAL IS RECEIVED */
     pause();

}

alarm_action()
{
/* PRINT THE VALUE OF THE ALARM COUNTER */
     printf("\t<%d\007>",number_of_alarms);

/* PASS SIGNAL AND FUNCTION TO signal */
     signal(SIGALRM,alarm_action);

/* SET ALARM CLOCK */
     alarm(SECONDS);

/* DECREMENT ALARM COUNTER */
     if (--number_of_alarms)
          pause();
}
```

Example 6-7: The signal, alarm, and pause Functions

6.11.120 sin

The function **sin** returns the sine of its radian argument. Both the argument and the sine value must be **double**.

- **Synopsis**

  ```
  #include math
  double sin(x)
  double x;
  ```

6.11.121 sinh

The function **sinh** returns the hyperbolic sine of its argument. Both the argument and the hyperbolic sine value must be **double**. The value of sinh(x), if it causes an overflow, is a **double** value with the largest possible magnitude and the appropriate sign.

- **Synopsis**

```
#include math
double sinh(x)
double x;
```

6.11.122 sleep

The function **sleep** suspends the execution of the current process for at least the number of seconds indicated by its argument. On success, **sleep** returns the number of seconds that the process slept. On error, **sleep** returns –1.

- **Synopsis**

```
int sleep(seconds)
unsigned seconds;
```

6.11.123 sprintf

See **printf**.

6.11.124 sqrt

The function **sqrt** returns the square root of its argument. The argument and the returned value are both **double**. **sqrt** returns 0 if x is negative, and **errno** is set to EDOM (defined in the **#include** module errno).

- **Synopsis**

```
#include math
double sqrt(x)
double x;
```

6.11.125 srand

See **rand**.

6.11.126 sscanf

See **scanf**.

6.11.127 ssignal

The function **ssignal** allows you to specify the action to be taken when a particular signal is raised. The first argument, sig, is a number or mnemonic associated with a signal. (The symbolic constants for signal values are defined in the **#include** module signal. See Table 6-12.) The second argument, action, represents the action to be taken when the signal is raised, or the address of a function that is executed when the signal is raised.

ssignal returns the address of the function previously established as the action for the signal. Note that the address may contain the value SIG_DFL (0) or SIG_IGN (1).

ssignal calls **signal** with the same arguments; the only difference between the two is in their return value on detecting an error (usually an invalid signal argument). **ssignal** returns 0 to indicate errors. For this reason, there is no way to know whether a return status of 0 indicates failure or whether it indicates that a previous action was SIG_DFL (0). **signal** returns -1 on error.

For more details on establishing actions for signals, see **signal**. For more details on the actions taken when a signal is raised, see **gsignal**.

- **Synopsis**

```
#include signal
/* ssignal IS A FUNCTION RETURNING
 * THE ADDRESS OF A FUNCTION RETURNING
 * AN INTEGER.
 */
int (*ssignal(sig,action)) ( )

/* action IS A POINTER TO A FUNCTION
 * RETURNING AN INTEGER.
 */
int sig, (*action)( );
```

6.11.128 strcat, strncat

The function **strcat** concatenates its second argument to the end of its first argument. Both arguments must be character strings, and, in the case of **strcat**, NUL-terminated.

The function **strncat** performs the same operation, but uses characters from the second argument up to a specified maximum unless the NUL terminator is encountered first. The argument max is an integer giving the maximum number of characters to use from string_2. If max is zero or negative, no characters are copied from string_2. If a **strncat** call reaches the specified maximum, **strncat** sets the next byte in string_1 to NUL.

Both functions return the address of the first argument, string_1. It is assumed to be large enough to hold the concatenated result.

- **Synopses**

```
char *strcat(string_1,string_2)
char *string_1,*string_2;

char *strncat(string_1,string_2,max)
char *string_1,*string_2;
int max;
```

6.11.129 strchr, strrchr

The function **strchr** returns the address of the first occurrence of a given character in a NUL-terminated string. It returns 0 if the character does not occur in the string. **strrchr** is similar, but returns the address of the last (rightmost) occurrence of the character.

- **Synopses**

```
char *strchr(string,character)
char *string,character;

char *strrchr(string,character)
char *string,character;
```

6.11.130 strcmp, strncmp

The function **strcmp** compares two ASCII character strings and returns a negative, zero, or positive integer, indicating that the first string is lexicographically less than, equal to, or greater than the second string. The returned value is obtained by subtracting the characters at the first position where the two strings disagree. (See Table G–1 for the numeric values of ASCII characters).

strncmp performs the same operation except that it compares a specified maximum number of characters in the two strings. The argument max gives the maximum number of characters (beginning with the first) to be compared. If max is zero or negative, no comparison is performed, and 0 is returned (the strings are considered equal).

With either function, the comparison is terminated when a NUL is encountered in one of the strings.

- **Synopses**

```
int strcmp(string_1,string_2)
char *string_1,*string_2;

int strncmp(string_1,string_2,max)
char *string_1,*string_2;
int max;
```

6.11.131 strcpy, strncpy

The function **strcpy** copies the argument string__2 into the argument string__1, stopping after string__2's NUL character is copied.

strncpy copies exactly max characters from string__2 to string__1; the value in string__2 is either truncated or padded with NUL characters. If string__2 is truncated, the copy in string__1 is not necessarily terminated by a NUL character.

Both functions return the address of string__1.

- **Synopses**

```
char *strcpy(string__1,string__2)
char *string__1,*string__2;

char *strncpy(string____1,string____2,max)
char *string__1,*string__2;
int max;
```

6.11.132 strcspn

The function **strcspn** searches a string for a character in a specified set of characters. It returns the number of characters that precede the matched one. (That is, the function spans the characters not in the set and returns the number of such leading characters. See also **strspn**.)

If the argument string is a null string, **strcspn** returns 0. If no characters match between string and charset, **strcspn** returns the length of string.

- **Synopsis**

```
int strcspn(string,charset)
char *string,*charset;
```

- **Example**

Example 6-8 shows how **strcspn** interprets four different kinds of arguments.

```
#include stdio
main()
{
     FILE *outfile;
     outfile = fopen("strcspn.out","w");
     fprintf(outfile,"strcspn with null charset: %d\n",
          strcspn("abcdef",""));
     fprintf(outfile,"strcspn with null string: %d\n",
          strcspn("","abcdef"));
     fprintf(outfile,"strcspn(abc,abc): %d\n",
          strcspn("abc","abc"));
     fprintf(outfile,"strcspn(abc,def): %d\n",
          strcspn("abc","def"));
}
```

The example writes the following to strcspn.out:

```
strcspn with null charset: 6
strcspn with null string: 0
strcspn(abc,abc): 0
strcspn(abc,def): 3
```

Example 6–8: The strcspn Function

6.11.133 strlen

The function **strlen** returns the length of a string of ASCII characters. The returned length does not include the terminating NUL character (\0).

- **Synopsis**

```
int strlen(string)
char *string;
```

6.11.134 strncat

See **strcat**.

6.11.135 strncmp

See **strcmp**.

6.11.136 strncpy

See **strcpy**.

6.11.137 strpbrk

The function **strpbrk** searches a string for the occurrence of one of a specified set of characters. It returns the address of the first character in the string that is in the set, or NULL if no character is in the set.

- **Synopsis**

```
char *strpbrk(string,charset)
char *string,*charset;
```

6.11.138 strrchr

See **strchr**.

6.11.139 strspn

The function **strspn** searches a string for the occurrence of a character that is not in a specified set of characters. It returns an **int** giving the number of characters that precede the mismatched character. (That is, the function spans the characters in the set and returns the number of such leading characters. See also **strcspn**.)

If charset is a null string, **strspn** returns 0. If all the characters in string are also in charset, the function returns the length of string.

- **Synopsis**

```
int strspn(string,charset)
char *string,*charset;
```

- **Example**

Example 6–9 shows how **strspn** interprets different arguments.

```
#include stdio
main()
{
     FILE *outfile;
     outfile = fopen("strspn.out","w");

     fprintf(outfile,"strspn with null charset: %d\n",
          strspn("abcdef",""));

     fprintf(outfile,"strspn with null string: %d\n",
          strspn("","abcdef"));

     fprintf(outfile,"strspn(abc,abc): %d\n",
          strspn("abc","abc"));

     fprintf(outfile,"strspn(abc,def): %d\n",
          strspn("abc","def"));

}
```

The example writes the following to strspn.out:

```
strspn with null charset: 0
strspn with null string: 0
strspn(abc,abc): 3
strspn(abc,def): 0
```

Example 6-9: The strspn Function

6.11.140 tan

The function **tan** returns a **double** value that is the tangent of its radian argument, which must also be **double**. The value of tan(x) at its "singular points" (...-3pi/2,-pi/2,pi/2...) is the largest possible **double** value, and **errno** is set to ERANGE.

- **Synopsis**

```
#include math
double tan(x)
double x;
```

6.11.141 tanh

The function **tanh** returns a **double** value that is the hyperbolic tangent of its **double** argument.

- **Synopsis**

```
#include math
double tanh(x)
double x;
```

6.11.142 time

The function **time** returns the time elapsed since 00:00:00, January 1, 1970, in seconds. If **time**'s argument is not null, it points to the place where the returned time is also stored.

- **Synopses**

```
long time(time_location)
long *time_location;
```

6.11.143 times

The function **times** returns the accumulated times of the current process and of its terminated child processes. The times are placed in a time structure defined below. For both process and children times, the structure breaks down the time by user and system time. Since VAX/VMS does not differentiate between system and user time, all system times are returned as 0. Accumulated CPU times are returned in 10-millisecond units.

```
struct tbuffer
    {
    int proc_user_time;
    int proc_system_time;
    int child_user_time;
    int child_system_time;
    };
```

- **Synopsis**

```
times(buffer)
struct tbuffer *buffer;
```

6.11.144 tmpfile

The function **tmpfile** creates a temporary file that is opened for update. The file exists only for the duration of the process and is preserved across forks. The function returns the address of a FILE structure (defined in the stdio module), or a null pointer value if there is an error.

- **Synopsis**

```
#include stdio
FILE *tmpfile( )
```

6.11.145 tmpnam

The function **tmpnam** creates a character string that can be used in place of the file-name argument in other function calls. If the name argument is null, **tmpnam** returns the address of an internal storage area. If name is not null, then it is taken to be the address of an area of length L_tmpnam (defined in the **#include** module stdio). In this case, name is returned by **tmpnam**. Successive calls to **tmpnam** with null arguments cause the current name to be overwritten.

- **Synopsis**

```
#include stdio
char *tmpnam(name)
char *name;
```

6.11.146 toascii

The function **toascii** converts its argument, an 8-bit ASCII character, to a 7-bit ASCII character. **toascii** is a macro.

- **Synopsis**

```
#include ctype
int toascii(character)
char character;
```

6.11.147 tolower, _tolower

The function **tolower** converts its argument, an uppercase alphabetic ASCII character, to lowercase. If the argument is already a lowercase character, it is returned unchanged. **_tolower** is implemented as a macro; **tolower** as a function.

- **Synopses**

```
char tolower(character)
char character;

#include ctype
char _tolower(character)
char character;
```

6.11.148 toupper, _toupper

The function **toupper** returns its argument, an ASCII lowercase alphabetic character, converted to uppercase. If the argument is already uppercase, it is returned unchanged. **_toupper** is implemented as a macro; **toupper** as a function.

- **Synopses**

```
char toupper(character)
char character;

#include ctype
char _toupper(character)
char character;
```

6.11.149 umask

The function **umask** creates a file protection mask that is used whenever a new file is created, and returns the old mask value. The actual file protection of a newly created file is the bitwise AND of the mode with the complement of the **umask** argument. The mode is supplied when the file is opened. The **umask** argument shows which bits to turn off when a new file is created. Initially, the mask is 0 (no restrictions).

See also **chmod**.

- **Synopsis**

```
int umask(mode_complement)
unsigned mode_complement;
```

6.11.150 ungetc

The function **ungetc** writes a character to the buffer of a file and leaves the file positioned before the character. The character is said to be "pushed back" onto the file, since it will be returned by the next **getc** call. The function returns the pushed-back character or EOF if it cannot push the character back.

One push-back is guaranteed, provided something has previously been read from the file. **fseek** erases all memory of pushed-back characters.

- **Synopsis**

```
#include stdio

int ungetc(character,file_pointer)
char character;
FILE *file_pointer;
```

6.11.151 vfork

The function **vfork** sets up the communication channels necessary to spawn and control a new process. The process from which **vfork** is called is known as the the parent process. The spawned process is known as the child process.

Note that **vfork** does not itself create a new process. Instead, it creates a duplicate of the caller's call frame, makes a record of the call's location (that is, a record of who the parent is), and returns a value. The return value is either 0 or the process ID of the child. That is, **vfork** returns 0 in the child process and a process ID in the parent.

You can use the value returned by **vfork** to control the parent and child processes; usually, the return value is used to decide whether to call one of the **exec** functions to create the child process. (See Example 6-10.) A child process created in such a manner inherits the following context from its parent:

- The user ID, group ID, and user name.
- All signals for which the action is to ignore the signal (see **signal**); signals that are caught in the parent are reset to their default handling in the child.
- The set of files opened by the parent. However, only record devices and files opened for reading can be shared by the parent and child, and the pointers to positions in the input files cannot be shared. Files opened by the parent for writing cannot be shared.
- The environment array of the parent (see **getenv** and the **exec** functions).

■ **Synopsis**

int vfork( )

■ **Example**

Example 6–10 shows how **vfork** is used to control the creation of a child process and to monitor its execution and return status. The child process executes the image in the file CHILD1.EXE.

```
#include stdio
#include ssdef

main( )
{
     int status,cstatus;

     /* SET UP CHILD PROCESS */
     if ((status = vfork( )) != 0)
          {
          /* WAIT FOR CHILD PROCESS TO RETURN */
          if ((status = wait(&status)) == -1)
               {
               perror("parent1");
               exit(SS$_NORMAL);
               }
          printf("Child's final status: %d\n",cstatus);
          }
     else
          {
          /* EXECUTE THE IMAGE IN CHILD1.EXE */
          if ((status = execl("child1","child_name",
                         "child_arg1",0)) == -1)
               {
               perror("parent1");
               exit(SS$_NORMAL);
               }
          }
}
```

Example 6–10: The vfork Function

6.11.152 wait

The function **wait** suspends the calling process until a signal is received or until one of its child processes terminates. If a child has terminated since the last **wait**, **wait** returns immediately. If there are no children, **wait** returns immediately with a return value of -1. A normal return value is the process ID of the terminated child.

The argument of **wait** points to an integer that receives the status with which the child was terminated. If the status is null, then the normal return value is returned (there are no side effects).

- **Synopsis**

```
int wait(status)
int *status;
```

6.11.153 write

The function **write** writes a specified number of bytes from a buffer to a file. The buffer argument is the address of n number of bytes of contiguous storage.

The function returns the number of bytes actually written. It returns -1 for errors, including undefined file descriptors, illegal buffer addresses, and physical I/O errors.

The file descriptor must refer to a file opened for writing or update (see **open**).

- **Synopsis**

```
int write(file_descriptor,buffer,nbytes)
int file_descriptor,nbytes;
char *buffer;
```

Chapter 7

Preprocessor Control Lines

Preprocessor control lines are lines in the source file that modify the action of the compiler.[1] The control lines are introduced by number signs (#) and do not end with semicolons. The number sign must appear in column 1. The effects of preprocessor control lines are independent of the usual scope rules and persist from their occurrence until the end of the compilation. Preprocessor control lines are not defined formally by the C language. Their implementations may differ from one compiler to another.

The control lines are:

- **#define** — defines token replacements (including preprocessor macro substitutions)
- **#include** — includes source text from an external file or library
- **#if, #ifdef, #ifndef, #else,** and **#endif** — control conditional compilation
- **#line** — specifies a line number
- **#module** — specifies a module name to the VAX/VMS Linker

7.1 Token Replacement

The **#define** control line specifies a token string that is substituted for every subsequent occurrence of the indicated identifier in the program text (unless it occurs inside a **char** constant, a comment, or a quoted string). The **#define** control line's forms are:

#define identifier *token-string*
#define identifier(identifier,...) *token-string*

If the token string is omitted, the identifier is deleted from the text given to the compiler.

---

1. The term *preprocessor* is used in this manual for consistency with other writing on the C language. However, VAX-11 C differs from other implementations in that these control lines are processed by an early phase of the compiler, not by a separate program.

After a token string is substituted in the source file, the compiler rescans the source file from the beginning to determine whether any previously replaced token strings contain identifiers defined by other **#define** control lines. If so, the identifiers are replaced by their token strings. For example:

```
/* SHOW MULTIPLE SUBSTITUTIONS AND LISTING FORMAT */
#define AUTHOR william + LAST

main()
{
   int writer,william,shakespeare,yeats;
#define LAST shakespeare
   writer = AUTHOR;

#define LAST yeats

   writer = AUTHOR;
}
```

When this text is compiled with the command

```
$ CC/SHOW=INTERMEDIATE RESCAN(RET)
```

the following listing results:

```
 1                /* SHOW MULTIPLE SUBSTITUTIONS AND
                     LISTING FORMAT */
 2                #define AUTHOR william + LAST
 3
 4                main()
 5                {
 6      1         int writer,william,shakespeare,yeats;
 7      1
 8      1         #define LAST shakespeare
 9      1
10      1         writer = AUTHOR;
              1   writer = william + LAST;
              2   writer = william + shakespeare;
11      1
12      1         #define LAST yeats
13      1
14      1         writer = AUTHOR;
              1   writer = william + LAST;
              2   writer = william + yeats;
15      1         }
```

Line 8 establishes shakespeare as the substitution for LAST, and line 2 establishes william + LAST as the substitution for AUTHOR. In line 10, AUTHOR is replaced by william + LAST. Then, the result is rescanned for other substitutable text, as a result of which LAST is replaced by shakespeare. There is no further substitution possible. In

line 12, the **#define** control line changes the substitution for LAST from shakespeare to yeats. In line 14, the final text becomes:

```
writer = william + yeats;
```

The **#define** control line may be continued onto subsequent lines if necessary. Each line to be continued must be terminated by a backslash (\) and a newline. The backslash and newline do not become part of the definition. The first character in the next line is logically adjacent to the character that immediately precedes the backslash. The backslash/newline as a continuation sequence is valid anywhere after the identifier being defined, or anywhere after the left parenthesis in a macro definition.

Comments can be continued without the backslash/newline. In the following example, all of the text must appear on the same line unless comments appear in the <white-space>:

```
#<white-space>define<white-space>identifier[(]
```

The optional left parenthesis begins a macro parameter list (see Section 7.1.2), and it *must not* be separated from the identifier.

7.1.1 Constant Identifiers

The first form of the **#define** control line defines a simple substitution, usually of a constant for a frequently used identifier. A common use of the control line is to define the end-of-file indicator:

```
#define EOF (-1)
```

The substitution text for this example is parenthesized to avoid lexical ambiguities when the text is substituted in the program, as in

```
i=EOF;
```

7.1.2 Macro Substitutions

Macros are defined with a **#define** control line of the form:

```
#define name([parm1[,parm2,...]]) [token-string]
```

where name, parm1, parm2, and so forth are identifiers, and token-string is arbitrary text.

Anywhere in the source file following such a control line (except within comments, character constants, or string constants), a macro reference of the form

```
name([arg1[,arg2,...]])
```

is replaced by the token string from the control line. Furthermore, any formal parameters (such as parm1) that appear in the token string are replaced by the corresponding arguments (such as arg1) from the reference.

For example, the definition of the macro **_toupper** is contained in the C definition module ctype. This file defines that macro in the following manner:

```
#define _toupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) & 0X5F : (c))
```

When you reference the macro **_toupper**, the compiler substitutes the macro definition and replaces the parameter (c) with the argument you give in the macro reference.

Preprocessor control lines and the macro reference have syntax that is independent of the C language. The following discussion gives the rules for specification of macro definitions and references:

1. *Macro Definitions.* The macro name and the formal parameters are identifiers and are specified according to the rules for identifiers in the C language.

   Spaces, tabs, and comments may be used freely within a **#define** control line. In particular, they may appear anywhere that the delta (△) symbol appears in the following example:

   ```
   #△define△name(△parm1△,△parm2△)△token-string△
   ```

   Note that white space cannot appear between the name and the left parenthesis that introduces the parameter list. White space may appear inside the token string. Also, at least one space, tab, or comment must separate the name from the word **define**. Comments may appear within the token string, but they do not become part of the macro definition.

2. *Macro References.* Comments and white-space characters (spaces, horizontal and vertical tabs, carriage returns, newlines, and form feeds) may be used freely within a macro reference. In particular, they may appear anywhere that the delta symbol appears in the following example:

   ```
   name△(△arg1△,△arg2△)
   ```

   Arguments consist of arbitrary text. Syntactically, they are not restricted to C expressions. They may contain embedded comments and white space. Comments are ignored, but the white space is preserved during the substitution.

   The number of arguments in the reference must match the number of parameters in the macro definition, although individual arguments may be null.

   Commas separate arguments except where they occur inside string or character constants, comments, or parentheses. Parentheses within arguments must be balanced.

When a macro reference is encountered in the source file, it is replaced by the token string from the macro definition. Then, the token string is scanned for the formal parameters. Any that are found are replaced by the corresponding actual arguments from the macro reference. Since the token string consists of arbitrary text, this replacement occurs even

if a parameter appears inside a character or string constant in the token string. (In order to be recognized, a parameter must be delimited from the surrounding text by white space or punctuation characters.)

You must be careful when specifying macro arguments that use the increment (++), decrement (--), and assignment (such as +=) operators or other arguments that may cause side effects (such as function calls). For example, you should not pass the following argument to the **_toupper** macro:

```
_toupper(*p++)
```

When *p++ is substituted in the macro definition, the result is:

```
((*p++) >= 'a' && (*p++) <= 'z' ? (*p++) & 0X5F : (*p++))
```

At run-time, p will have a different value at each reference to it.

If the token string is omitted from the macro definition, the entire macro reference simply disappears from the source text.

The token string in the macro definition, as well as actual arguments in a macro reference, may contain other macro references. Substitution occurs as expected, but such nested references are limited to a depth of 64. The maximum number of parameters or arguments is also 64.

7.1.3 Listing of Substituted Lines

The VAX-11 C compiler command has two /SHOW qualifiers that enable the listing of all lines that have been modified by macro substitutions.

With the qualifier

```
/SHOW=EXPANSION
```

the listing produced by the compiler shows the final form of a line (after all substitutions), preceded by its original form and followed by the listing of machine code, if any. Substituted lines are flagged in the margin.

With the qualifier

```
/SHOW=INTERMEDIATE
```

all intermediate substitutions are also listed, with one substitution per line.

Without one of these two qualifiers, only the original form of a line (before the substitutions) is listed.

The example in Section 7.1 demonstrates the effect of the /SHOW=INTERMEDIATE qualifier. For details on the format of VAX-11 C listings, see Appendix D.

7.1.4 Canceling Definitions

The control line

#undef identifier

cancels a previous definition of the identifier.

7.2 File Inclusion

The **#include** control line inserts external text into the token stream delivered to the compiler. Its forms are:

#include <file-spec>
#include "file-spec"
#include module-name

file-spec is a valid VMS file specification or logical name. module-name is the name of a module in a text library.

If the file-spec is delimited by angle brackets (<>), SYS$LIBRARY is searched for a file of that name.

If the file-spec is delimited by quotation marks (" "), and if, after applying the usual RMS defaults, no directory is known, then the directory containing the source file is searched for the file.

Any text not delimited in the **#include** control line is assumed to be the name of a module in a VAX/VMS text library. VAX-11 C text libraries are specified and searched as follows:

- A text library can be created with the LIBRARY command and specified with a qualifier on the VAX-11 C compile command.
- If more than one compilation is done by a single compile command, the library must be specified for each source file, as in:

```
CC sourcea+mylib/LIBRARY,sourceb+mylib/LIBRARY
```

- If more than one library is specified in the compile command, the libraries are searched in the specified order each time an **#include** control line is encountered. For example:

```
CC sourcea+mylib/LIBRARY+yourlib/LIBRARY
```

  In this example, references to **#include** modules are searched for first in MYLIB.TLB and then in YOURLIB.TLB.

- If no library is specified in the compile command, or if the specified module cannot be found in any of the specified libraries, the following actions are taken:

  — If the user has defined an equivalence name for C$LIBRARY that names a text library, that library is searched.

  — Any remaining unresolved module names are searched for in SYS$LIBRARY:CSYSDEF.TLB.

#include control lines may be nested to a depth of four.

7.3 Conditional Compilation

Five control lines are available to control conditional compilation. They delimit blocks of statements that are compiled if a certain condition is true; they may be nested. The beginning of the block of statements is marked by one of three control lines: **#if, #ifdef,** or **#ifndef.** Optionally, an alternative block of statements can be set aside with the **#else** control line. The end of the block is marked by an **#endif** control line.

If the condition checked by **#if, #ifdef,** or **#ifndef** is true, then all lines between an **#else** and **#endif** are ignored by the compiler. If the condition is false, then the lines between the **#if, #ifdef,** or **#ifndef** and an **#else** or **#endif** control line are ignored. Ignored lines are flagged with an X in the compiler listing margin.

The **#if** control line has the form:

```
#if constant-expression
```

It checks whether the constant expression is nonzero (true). The operands must be constants. The increment (++), decrement (--), **sizeof**, pointer (*), address (&), and cast operators are not allowed in the constant expression.

The constant expression in an **#if** control line is subject to text replacement and can, therefore, contain references to identifiers defined in previous **#define** control lines. The replacement occurs before the expression is evaluated.

If an identifier used in the expression is not currently defined, the compiler issues a warning message (UNDEFIFMAC), and treats the identifier as though it were the constant zero.

The **#ifdef** control line has the form:

```
#ifdef identifier
```

It checks whether the identifier was previously defined by a **#define** control line.

The **#ifndef** control line has the form:

```
#ifndef identifier
```

It checks to see if the identifier is not defined or if it has been undefined by the **#undef** control line.

The **#else** control line has the form:

```
#else
```

It delimits alternative source lines to be compiled if the condition tested for in the corresponding **#if, #ifdef,** or **#ifndef** control line is false. An **#else** control line is optional.

The **#endif** control line has the form:

```
#endif
```

It ends the scope of the above control lines.

The VAX-11 C compiler defines three preprocessor substitutions with the names vax, vms, and vax11c. These symbols are defined as if the following text fragment were included by the compiler before every compilation source group.

```
#define vms     1
#define vax     1
#define vax11c  1
```

The symbols may be used by the C programmer to conditionally compile C programs used on more than one operating system to take advantage of system-specific features. For example:

```
    .
    .
#if vax11c
#include rms      /* include RMS definitions */
#endif
    .
    .
```

Because the VAX-11 C compiler will substitute 1 for every occurrence of these identifiers in a program, these three identifiers should be considered reserved by DIGITAL. The effect of these predefinitions may be removed by explicitly undefining the conflicting name.

7.4 Specification of Line Numbers

The C compiler keeps track of information about relative line numbers in each file involved in the compilation. When it displays a diagnostic message at the terminal, the compiler also displays this information. The **#line** control line specifies a new starting line number (the number is incremented from that point) and a new identifier or string for the file containing the control line. The new information is in effect until the end of the file or until another **#line** control line changes it.

The formats of the **#line** control line are:

#line constant identifier
#line constant string
constant identifier
constant string

The constant must be an unsigned decimal integer. It supplies the line number. The second parameter can be specified as either a VAX-11 C identifier or a character-string constant. It supplies the VAX/VMS file specification. The character string must not exceed 128 characters.

7.5 Specification of Module Name and Identification

The **#module** control line causes the following information to be applied to the object module of the current compilation:

- A module name that appears in the compiler listing file and linker load map. The module name is then recognized by the debugger and librarian.
- An optional identification string (such as a version number) which also appears in the listing file and load map.

If it does not encounter a **#module** control line during compilation, the VAX-11 C compiler gives the output object file (if any) the same module name as the first function it encounters and it gives the file an identification of V1.0. You can use the **#module** control line to override these defaults.

The formats of the control line are:

```
# module identifier identifier
# module identifier string
```

The first parameter must be a valid VAX-11 C identifier. It specifies the module name to be used by the linker. The second parameter specifies the optional identification that appears on listings and in the object file. It must be either a valid VAX-11 C identifier or a character-string constant with no more than 31 characters.

Only one **#module** line can be processed per compilation, and that line must appear before any C language text. (That is, it can follow other control lines, such as **#define**, but it must precede any function definitions or external data definitions.)

The parameters in a **#module** line are subject to text replacement and can, therefore, contain references to identifiers defined in previous **#define** control lines. The replacement occurs before the parameters are processed.

The **#module** control line is a VAX/VMS extension of the preprocessor; it may not occur in other C implementations.

Chapter 8

Using VAX–11 Record Management Services (RMS)

The C programming language does not provide built-in, or predefined, facilities for handling files. All input and output (I/O) operations in a C program are performed by calls to external functions, which vary in nature and number from one implementation to another. The file-handling capabilities of VAX–11 C fall into two distinct categories:

1. Those functions which follow the conventions that allow programs written in one version of the language to run with little or no modification with other versions of the language and run-time library. This involves the use of stream files and C functions that perform standard and UNIX I/O (refer to Chapter 6).
2. The RMS functions which are not portable to other C implementations, but which provide more kinds of file organization and more record access modes.

This chapter briefly reviews the basic concepts and facilities of VAX–11 RMS and shows examples of their application in VAX–11 C programming. Because this is an overview, not all RMS concepts and features are explained. If you want more complete information, consult the following manuals in the VAX/VMS document set:

- *Introduction to VAX–11 Record Management Services.* This document contains a general description of the record management services of the VAX/VMS operating system. The information in this document is introductory; programming techniques are not presented. Indexed sequential access is described.
- *VAX–11 Record Management Services Reference Manual.* This manual fully describes the user interface to RMS. It includes some introductory information on RMS programming and detailed definitions of all RMS access blocks, attribute blocks, and macro instructions.

8.1 RMS File Organization

VAX-11 RMS supports three kinds of file organization:

- Sequential organization
- Relative organization
- Indexed organization

The organization of a file determines the way the file is stored on the medium and, consequently, the possible operations on records. A file's organization is specified when the file is created, and it cannot be changed.

8.1.1 Sequential Organization

Sequential files have consecutive records. There are no empty records separating records that contain data. This organization limits the operations on the file to:

- Positioning the file at a particular record, generally by sequentially moving from one record to the next.[1]
- Reading data from any record.
- Writing data by adding records at the end of the file.

Sequential organization is the only kind permitted for magnetic tape files.

8.1.2 Relative Organization

Relative files have records that occupy numbered, fixed-length cells. The records themselves need not have the same length. Cells can be empty or can contain records. Consequently, the following operations are permitted:

- Positioning the file at a particular record, usually by direct access. In direct access, the relative record number (actually the number of a cell) is usually used as a key to locate the cell and its record, without reference to other cells. The records can also be accessed sequentially, in which case any empty cells are ignored, or they can be accessed directly by record file address (RFA). (The RFA is returned in a parameter block whenever a record is written and can be used subsequently to locate the record directly.)

---

1. Direct access is also possible, either by key (relative record number) or by the record file address (RFA). However, access by RFA is limited to files on disk devices, and access by key is limited to disk files that also have fixed-length records. These access modes are unusual because most application programs do not keep track of record positions in sequential files.

- Reading a record from any cell.
- Deleting a record from any cell.
- Writing a record into any cell.

The relative file organization is possible only on disk devices.

8.1.3 Indexed Organization

Indexed files have records that contain, in addition to data and carriage-control information, one or more keys. Keys can be character strings, packed decimal numbers, and 16- or 32-bit signed or unsigned integers. Every record has at least one key, the primary key, whose value is fixed. Optionally, each record can have one or more alternate keys, whose values can vary.

Unlike relative record numbers used in relative files, key values in indexed files are not necessarily unique. When you create a file, you can specify that a particular key may have the same value in different records (these keys are called duplicate keys). Keys are defined for the entire file, in terms of their position within a record and their length.

In addition to maintaining its records, RMS builds and maintains indexes for each of the defined keys. As records are written to the file, their key values are inserted in order of ascending value in the appropriate indexes. This organization makes possible the following operations:

- Positioning the file at a particular record, by direct access. In direct access reads, either a primary or alternate key, plus a specified key value, is used to locate the record. In direct access writes (given a record that contains key values in the predefined positions), RMS automatically adds the record to the file and adds the primary and alternate key values to the appropriate indexes. Records can also be accessed sequentially, where the sequence is defined by the index for a specified key. Finally, records can be accessed directly by RFA. (The RFA is returned in a parameter block whenever a record is written and can be used subsequently to locate the record directly.)
- Reading any record, including sequential reads controlled by a key's index.
- Deleting any record.
- Updating an alternate key's value, if the key's definition permits its value to change.
- Writing records selectively, based on the value of a key and, when allowed in the key's definition, based on duplicate values. If duplicate values are permitted, you can write records containing key values that are already present in the key's index. If duplicate values are not permitted, such write operations are rejected.

Indexed organization is possible only on disk devices.

8.2 Record Access Modes

The record access modes, summarized in Section 8.1, are sequential, direct (random) by key, and direct by record file address. Again, the direct access modes are possible only with files that reside on disks.

Unlike a file's organization, the record access mode is not a permanent attribute of the file. During the processing of a file, you can switch from one access mode to any other permitted for that file organization. For example, indexed files are often processed by locating a record directly by key, and then using that key's index to read sequentially all the indexed records (this is sometimes called the indexed sequential access method, or ISAM).

8.3 RMS Record Formats

Records in RMS files can have the following formats:

- Fixed-length format, where the length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format, where the maximum length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format with a fixed-length control area (VFC), where every record is prefixed by a fixed-length field. This format is permitted only with sequential and relative files.

8.4 RMS Functions

RMS provides a number of functions that create and manipulate files. These functions use RMS data structures to define the characteristics of a file and its records.

The RMS data structures are grouped into four main categories, as follows:

- File Access Block (FAB). Defines the file's characteristics, such as file organization and record format.
- Record Access Block (RAB). Defines the way in which records are processed, such as the record access mode.
- Extended Attribute Blocks (XAB). Various kinds of extended attribute blocks contain additional file characteristics, such as the definition of keys in an indexed file. (Extended attribute blocks are optional.)
- Name Block (NAM). Defines all or part of a file specification to be used when an incomplete file specification is given in an OPEN or CREATE operation. (Name blocks are optional.)

RMS uses these data structures to perform file and record operations. Table 8-1 lists some of the commonly used functions.

Table 8–1: Common RMS Run-Time Processing Functions

Category	Function	Description
File Processing	sys$create	Creates and opens a new file of any organization
	sys$open	Opens an existing file and initiates file processing
	sys$close	Terminates file processing and closes the file
	sys$erase	Deletes a file
Record Processing	sys$connect	Associates a file access block with a record access block to establish a record access stream; a call to this function is required before any other record processing function can be used
	sys$get	Retrieves a record from a file
	sys$put	Writes a new record to a file
	sys$update	Rewrites an existing record to a file
	sys$delete	Deletes a record from a file
	sys$rewind	Positions the record pointer to the first record in the file
	sys$disconnect	Disconnects a record access stream

All RMS functions are directly accessible from VAX-11 C programs. The synopsis for any RMS function has the following form:

```
int sys$name(pointer)
struct rms_structure *pointer;
```

In this synopsis, name corresponds to the name of the RMS function (such as OPEN or CREATE); rms_structure corresponds to the name of the structure being used by the function.

The file-processing functions require a pointer to a file access block as an argument; the record-processing functions require a pointer to a record access block as an argument. For example, because **sys$create** is a file-processing function, its synopsis would be as follows:

```
int sys$create(fab)
struct FAB *fab;
```

Note that these synopses do not show all the options available when an RMS function is invoked. Refer to Chapter 8 of the *VAX-11 Record Management Services Reference Manual* for a complete description of the RMS calling sequence.

Finally, all of the RMS functions return an integer status value. The format of RMS status values follows the standard format described in Chapter 9 of this manual. Because they return a 32-bit integer, you do not need to declare the RMS functions before you use them.

8.5 Writing VAX-11 C Programs Using RMS

VAX-11 C supplies a number of **#include** modules that describe the RMS data structures and status codes. These modules are listed in Table 8-2.

Table 8-2: VAX-11 C RMS #include Modules

Module Name	Structure Tag(s)	Description
fab	FAB	Defines the file access block structure
rab	RAB	Defines the record access block structure
nam	NAM	Defines the name block structure
xab	XAB	Defines the extended attribute block structure
rmsdef	—	Defines the completion status codes that RMS returns after every file- or record-processing operation
rms	all tags	Includes all of the above modules

Most VAX-11 C programmers simply include the rms module, which includes all the other modules.

These **#include** modules define all the data structures as structure tag names. However, they perform no allocation or initialization of the structures; these modules describe only a template for the structures. To use the structures, you must create storage for them and initialize all the structure members as required by RMS.

To assist in the initialization process, VAX-11 C provides initialized RMS data structure prototypes. You can copy these **readonly** prototypes to your uninitialized structure definitions with a structure assignment. You can choose to take the default values for each of the structure members (as initialized by the prototypes), or you can tailor the contents of the structures to fit your requirements. In either case, you must use the templates to allocate storage for the structure and to define the members of the structure.

The initialized prototypes supply the RMS default values for each member in the structure; they specify none of the optional parameters. To determine what default values are supplied by the prototypes, consult the *VAX-11 Record Management Services Reference Manual.*

The prototype data structures, and the structures which they initialize, are listed in Table 8-3.

Table 8-3: RMS Prototype Data Structures

Prototype	Structure Tag	Initialized Structure
cc$rms_fab	FAB	File access block
cc$rms_rab	RAB	Record access block
cc$rms_nam	NAM	Name block
cc$rms_xaball	XAB	Allocation extended attribute block
cc$rms_xabdat	XAB	Date and time extended attribute block
cc$rms_xabfhc	XAB	File header characteristics extended attribute block
cc$rms_xabkey	XAB	Indexed file key extended attribute block
cc$rms_xabpro	XAB	Protection extended attribute block
cc$rms_xabrdt	XAB	Revision date and time extended attribute block
cc$rms_xabsum	XAB	Summary extended attribute block

You need not declare these structures before referencing them; the declarations of these structures are contained in the appropriate **#include** module.

The names of the structure members conform to the following RMS naming convention:

```
typ$s_fld
```

where typ is the abbreviation for the structure, s is the size of the member (such as l for longword or b for byte), and fld is the member name (such as sts for the completion status code). See the *VAX-11 Record Management Services Reference Manual* for a description of the members in each structure.

8.5.1 Initializing File Access Blocks

The file access block defines the attributes of the file. To initialize a file access block, you assign the values in the initialized data structure cc$rms_fab to the address of the file access block defined in your program. For example:

```
/* DECLARE ALL RMS DATA STRUCTURES */
#include rms

/* DEFINE A FILE ACCESS BLOCK */
struct FAB fblock;
main()
{
/* INITIALIZE THE STRUCTURE */
     fblock = cc$rms_fab;
}
```

Any of these RMS structures may be dynamically allocated. For example, another way to allocate a file access block is as follows:

```
/* DECLARE ALL RMS DATA STRUCTURES */
#include rms
main()
{
     /* ALLOCATE DYNAMIC STORAGE */
     struct FAB *fptr = malloc(sizeof (struct FAB));

     /* INITIALIZE THE STRUCTURE */
     *fptr = cc$rms_fab;
}
```

More often than not, you will want to change the default values supplied by the prototype. If so, you must reinitialize the members of the structure individually. You initialize a member by giving the offset of the member and assigning a value to it. For example, the statement

```
fblock.fab$l_xab = &primary_key;
```

assigns the address of the extended attribute block named primary_key to the fab$l_xab member of the file access block named fblock.

8.5.2 Initializing Record Access Blocks

The record access block specifies how records are processed. You initialize a record access block in a manner similar to the way in which you initialize a file access block. For example:

```
#include rms
struct FAB fblock;

/* DEFINE A RECORD ACCESS BLOCK */
struct RAB rblock;
main()
{
     /* INITIALIZE THE STRUCTURE */
     fblock = cc$rms_fab;
     rblock = cc$rms_rab;

     /* INITIALIZE THE fab MEMBER */
     rblock.rab$l_fab = &fblock;
}
```

8.5.3 Initializing Extended Attribute Blocks

There is only one extended attribute block structure, but you can initialize it seven ways. The extended attribute blocks define additional file attributes that are not defined elsewhere. For example, the key extended attribute block is used to define the keys of an indexed file.

All extended attribute blocks are "chained" off of a file access block in the following manner:

1. In a file access block, you initialize the fab$l_xab field with the address of the first extended attribute block.
2. You designate the next extended attribute block in the chain in the xab$l_nxt field of any subsequent extended attribute blocks. You chain each subsequent extended attribute block in order by the key of reference (first the primary key, then the first alternate key, then the second alternate key, and so on).
3. You initialize the xab$l_nxt member of the last extended attribute block in the chain with 0 (the default), to indicate the end of the chain.

You go through the same steps to declare extended attribute blocks as you would to declare the other RMS data structures:

1. You define the structures with **#include** modules.
2. You assign a specific prototype to the structure in your program.
3. You initialize the members of the structure with the desired values.

In the following example, two extended attribute block structures are declared. They are initialized as key extended attribute blocks with the cc$rms_xabkey prototype. The xab$l_nxt member of the primary key is initialized with the address of the alternate_key extended attribute block.

```
#include rms
struct XAB primary_key,alternate_key;

main()
{
    primary_key = cc$rms_xabkey;
    alternate_key = cc$rms_xabkey;
    primary_key.xab$l_nxt = &alternate_key;
}
```

8.5.4 Initializing Name Blocks

The name block contains default file name values, such as the directory or device specification, file name, or file type. If you do not specify one of the parts of the file specification when you open the file, RMS uses the values in the name block to complete the file specification and places the complete file specification in an array.

You create and initialize name blocks in the same manner as you would initialize the other RMS data structures; for example:

```
#include rms

struct NAM nam;
struct FAB fab;

main()
{
      fab = cc$rms_fab;
      nam = cc$rms_nam;

/* DEFINE THE ARRAY WHERE THE EXPANDED
 * FILE SPECIFICATION WILL BE PLACED
 */
      char expanded_name[NAM$C_MAXRSS];

/* INITIALIZE THE APPROPRIATE MEMBERS */
     fab.fab$l_nam = &nam;
     nam.nam$l_esa = &expanded_name;
     nam.nam$b_ess = sizeof expanded_name;
}
```

8.6 RMS Example Program

The example program in this section uses RMS functions to maintain a simple employee file. The file is an indexed file with two keys: social security number and last name. The fields in the record are character strings defined in a structure with the tag record.

The records have the carriage-return attribute. Individual fields in each record are padded with blanks for two reasons. First, those fields that are key fields must be padded in some way; RMS does not understand C-style strings with the trailing NUL character. Second, the choice of blank padding as opposed to NUL padding allows the file to be printed or typed without conversion.

The program does not perform range or bounds checking. Only the error checking that shows the mapping of VAX-11 C to RMS is performed. Any other errors are considered to be fatal.

The program is divided into the following sections:

- External data declarations and definitions
- Main program section
- Function to initialize the RMS data structures
- Internal functions to open the file, display HELP information, pad the records, and process fatal errors
- Utility functions:
  — add
  — delete
  — type
  — print
  — update

To run this program, you would go through the following steps:

1. Create a source file. (The name of the source file in this example is RMSEXP.C.) Chapter 13 describes the EDT editor which could be used to create the source file.
2. Compile the source file with the command

   ```
   $ CC RMSEXP
   ```

   Chapter 14 describes the compile command and its options.
3. Link the program with the command

   ```
   $ LINK RMSEXP,SYS$LIBRARY:CRTLIB/OLB
   ```

   Chapter 14 describes the link command.
4. Because the program expects command line arguments, it must be defined as a foreign command, as follows:

   ```
   $ RMSEXP :== $device:[directory]RMSEXP
   ```

   where device is the logical or physical name of the device containing your directory; directory is the name of your directory. The device name must be preceded by the dollar sign ($) to be recognized as a foreign command by the DCL interpreter.

   Chapter 14 describes the use of foreign commands to execute a program.
5. Run the program, using the foreign command.

   ```
   $ RMSEXP filename
   ```

The complete listing (by section) of the example program follows. Notes on each section are keyed to the numbers on the listing.

Example 8-1 shows the external data declarations and definitions.

❶ The rms module defines the RMS data structures. The stdio module contains the standard I/O definitions. The ssdef module contains the system services definitions.

❷ Preprocessor variables and macros are defined. A default file-name string is defined as ".dat."

The sizes of the fields in the record are also defined. Some (such as the social security number field) are given a constant length. Others (such as the record size) are defined as macros; the size of the field is determined with the **sizeof** operator. VAX-11 C evaluates constant expressions, such as KEY__SIZE, at compile time. No special code is necessary to calculate this value.

❸ Static storage for the RMS data structures is declared. The file access block, record access block, and extended attribute blocks are defined by the rms module. One extended attribute block is defined for the primary key and one is defined for the alternate key.

❹ The records in the file are defined using a structure with four fields of character arrays.

❺ The BUFSIZ constant is used to define the size of the array that will be used to buffer input from the terminal. The filename variable is defined as a pointer to **char.**

❻ rms__status is a variable that is used to receive RMS return status information. After each function call, RMS returns status information as an integer. This return status is used to check for specific errors, end-of-file, or successful program execution.

```
❶ #include rms
  #include stdio
  #include ssdef

❷ #define DEFAULT_FILE_NAME        ".dat"

  #define RECORD_SIZE              (sizeof record)
  #define SIZE_SSN                 15
  #define SIZE_LNAME               25
  #define SIZE_FNAME               25
  #define SIZE_COMMENTS            15
  #define KEY_SIZE                 (SIZE_SSN > SIZE_LNAME ? SIZE_SSN : SIZE_LNAME)

❸ struct FAB fab;
  struct RAB rab;
  struct XAB primary_key,alternate_key;

❹ struct {
          char     ssn[SIZE_SSN],last_name[SIZE_LNAME];
          char     first_name[SIZE_FNAME], comments[SIZE_COMMENTS];
          }
          record;

❺ char response[BUFSIZ],*filename;

❻ int rms_status;
```

Example 8-1: External Data Declarations and Definitions

The main function, shown in Example 8-2, controls the general flow of the program.

❶ The main function is entered with two parameters. The first is the number of arguments used to call the program; the second is a pointer to the first argument (filename).

❷ This statement checks to see if the correct number of arguments were used when the program was called.

❸ If the file name is included in the command line to execute the program, that file name is used. (Note that if a file type is not given, .dat is the type.) If no file name is specified, then the file name is personnel.dat.

❹ The file access block, record access block, and extended attribute blocks are initialized.

❺ The file is opened using the RMS **sys$open** function.

❻ The program displays a menu and checks for end-of-file (CTRL/Z).

❼ A **switch** statement and a set of **case** statements control the function to be called, determined by the response from the terminal.

❽ The program ends when CTRL/Z is entered in response to the menu. At that time, the RMS **sys$close** function closes the employee file.

❾ The rms__status variable is checked for a return status of RMS$__NORMAL. If the file is not closed successfully, then the error-handling function terminates the program.

```
❶ main(argc,argv) char **argv;
  {

❷   if (argc < 1 || argc > 2)
        printf("RMSEXP - incorrect number of arguments");
    else
        {


        printf("RMSEXP - Personnel Database Manipulation Example\n");

❸       filename = (argc == 2 ? *++argv : "personnel.dat");
❹       initialize(filename);
❺       open_file();

        for(;;)
            {
❻           printf("\nEnter option (A,D,P,T,U) or ? for help :");
            gets(response);
            if (feof(stdin))
                break;
            printf("\n\n");

❼           switch(response[0])
                {
                case 'a': case 'A':     add_employee();
                                        break;
```

Example 8-2: Main Program Section

```
            case 'd': case 'D':     delete_employee();
                                    break;

            case 'p': case 'P':     print_employees();
                                    break;

            case 't': case 'T':     type_employees();
                                    break;

            case 'u': case 'U':     update_employee();
                                    break;

            default:                printf("RMSEXP - Unknown Operation.\n");
            case '?': case '\0':

                                    type_options();
                                    break;
            }
        }

❽   rms_status = sys$close(&fab);

❾   if (rms_status != RMS$_NORMAL)
        error_exit("$CLOSE");
    }
}
```

Example 8-2: (Cont.) Main Program Section

Example 8-3 shows the function that initializes the RMS data structures. Refer to the RMS documentation for more information about the file access block, record access block, and extended attribute block structure members.

❶ cc$rms_fab initializes the file access block with default values. Some members have no default values; they must always be initialized. (Such members include the file-name string address and size.) Other members can be initialized to override the default values.

❷ This statement initializes the file-processing options member with the "create-if" option. A file will be created if one does not exist.

❸ This statement initializes the record attributes member with the carriage-return control attribute. Records will be terminated with a carriage return/line feed when they are printed on the printer or displayed at the terminal.

❹ cc$rms_rab initializes the record access block with the default values. In this case, the only member that must be initialized is the rab$l_fab member, which associates a file access block with a record access block.

❺ cc$rms_xabkey initializes an extended attribute block for one key of an indexed file.

❻ The position of the key is specified by subtracting the offset of the member from the base of the structure.

❼ A separate extended attribute block is initialized for the alternate key.

❽ This statement specifies that more than one alternate key can contain the same value (XAB$M_DUP) and that the value of the alternate key can be changed (XAB$M_CHG).

❾ The key-name member is padded with blanks because it is a fixed-length 32-character field.

```
initialize(fn)  char    *fn;
{

❶   fab = cc$rms_fab;                   /* Start by initializing FAB */

    fab.fab$b_bks = 4;
    fab.fab$l_dna = DEFAULT_FILE_NAME;
    fab.fab$b_dns = sizeof DEFAULT_FILE_NAME -1;
    fab.fab$b_fac = FAB$M_DEL | FAB$M_GET |
                      FAB$M_PUT | FAB$M_UPD;
    fab.fab$l_fna = fn;
    fab.fab$b_fns = strlen(fn);
❷   fab.fab$l_fop = FAB$M_CIF;
    fab.fab$w_mrs = RECORD_SIZE;
    fab.fab$b_org = FAB$C_IDX;
❸   fab.fab$b_rat = FAB$M_CR;
    fab.fab$b_rfm = FAB$C_FIX;
    fab.fab$b_shr = FAB$M_NIL;
    fab.fab$l_xab = &primary_key;

❹   rab = cc$rms_rab;                   /* Initialize RAB */

    rab.rab$l_fab = &fab;

❺   primary_key = cc$rms_xabkey;        /* Initialize Primary Key XAB */
```

```
    primary_key.xab$b_dtp = XAB$C_STG;
    primary_key.xab$b_flg = 0;
❻   primary_key.xab$w_pos0 = (char *) &record.ssn - (char *) &record;
    primary_key.xab$b_ref = 0;
    primary_key.xab$b_siz0 = SIZE_SSN;
    primary_key.xab$l_nxt = &alternate_key;
    primary_key.xab$l_knm = "Employee Social Security Number       " ;

❼   alternate_key = cc$rms_xabkey;  /* Initialize Alternate Key XAB */

    alternate_key.xab$b_dtp = XAB$C_STG;
❽   alternate_key.xab$b_flg = XAB$M_DUP | XAB$M_CHG;
    alternate_key.xab$w_pos0 = (char *) &record.last_name - (char *) &record;
    alternate_key.xab$b_ref = 1;
    alternate_key.xab$b_siz0 = SIZE_LNAME;
❾   alternate_key.xab$l_knm = "Employee Last Name                 ";

}
```

Example 8-3: Function Initializing RMS Data Structures

Example 8-4 shows the internal functions for the program.

❶ The open_file function uses the RMS **sys$create** function to open the file, giving the address of the file access block as an argument. The function returns status information to the rms_status variable.

❷ The RMS **sys$connect** function associates the record access block with the file access block.

❸ The type_options function, called from the main function, prints help information. Once the help information is displayed, control returns to the main function, which processes the response that is typed at the terminal.

❹ For each field in the record, the pad_record function fills the remaining bytes in the field with blanks.

❺ This function handles fatal errors. It prints the function that caused the error, returns a VAX/VMS error code (if appropriate), and exits the program.

```
open_file()
{
    ❶   rms_status = sys$create(&fab);
        if (rms_status != RMS$_NORMAL && rms_status != RMS$_CREATED)
                error_exit("$CREATE");

        if (rms_status == RMS$_CREATED)
                printf("[Created new data file.]\n");

    ❷   rms_status = sys$connect(&rab);
        if (rms_status != RMS$_NORMAL)
                error_exit("$CONNECT");
}


❸ type_options()
{
        printf("Enter one of the following:\n\n");
        printf("A       Add an employee.\n");
        printf("D       Delete an employee specified by SSN.\n");
        printf("P       Print employee(s) by ascending SSN on line printer.\n");
        printf("T       Type employee(s) by ascending last name on terminal.\n");
        printf("U       Update employee specified by SSN.\n\n");
        printf("?       Type this text.\n");
        printf("^Z      Exit this program.\n\n");
}
```

Example 8-4: Internal Functions

❹
```
pad_record()
{
        int     i;

        for(i = strlen(record.ssn); i < SIZE_SSN; i++)
                record.ssn[i] = ' ';
        for(i = strlen(record.last_name); i < SIZE_LNAME; i++)
                record.last_name[i] = ' ';
        for(i = strlen(record.first_name); i < SIZE_FNAME; i++)
                record.first_name[i] = ' ';
        for(i = strlen(record.comments); i < SIZE_COMMENTS; i++)
                record.comments[i] = ' ';
}
```

❺
```
error_exit(operation) char *operation  /* Fatal error processing subroutine */
{
        printf("RMSEXP-file %s failed (%s)\n",operation,filename);
        exit(rms_status);
}
```

Example 8-4: (Cont.) Internal Functions

Example 8-5 shows the function that adds a record to the file. This function is called when 'a' or 'A' is entered in response to the menu.

❶ A series of **do** loops controls the input of information. For each field in the record, a prompt is displayed. The response is buffered and the field is copied to the structure.

❷ When all fields have been entered, the pad_record function pads each field with blanks.

❸ Three members in the record access block are initialized prior to writing the record. The record access member (rab$b_rac) is initialized for keyed access. The record buffer and size members (rab$l_rbf and rab$w_rsz) are initialized with the address and size of the record to be written.

❹ The RMS **sys$put** function writes the record to the file.

❺ The rms_status variable is checked. If the return status is normal, or if the record has a duplicate key value and duplicates are allowed, the function prints a message stating that the record was added to the file. Any other return value is treated as a fatal error.

```
add_employee()
{
❶   do
            {
            printf("(ADD)    Enter Social Security Number          ");
            gets(response);
            } while(strlen(response) == 0);

    strncpy(record.ssn,response,SIZE_SSN);

    do
            {
            printf("(ADD)    Enter Last Name                       ");
            gets(response);
            } while(strlen(response) == 0);

    strncpy(record.last_name,response,SIZE_LNAME);

    do
            {
            printf("(ADD)    Enter First Name                      ");
            gets(response);
            } while(strlen(response) == 0);
```

```
        strncpy(record.first_name,response,SIZE_FNAME);

        do
                {
                printf("(ADD)    Enter Comments                          ");
                gets(response);
                } while(strlen(response) == 0);

        strncpy(record.comments,response,SIZE_COMMENTS);

❷       pad_record();

❸       rab.rab$b_rac = RAB$C_KEY;
        rab.rab$l_rbf = &record;
        rab.rab$w_rsz = RECORD_SIZE;

❹       rms_status = sys$put(&rab);

❺       if (rms_status != RMS$_NORMAL && rms_status != RMS$_DUP && rms_status != RMS$_OK_DUP)
                error_exit("$PUT");
        else if (rms_status == RMS$_NORMAL || rms_status == RMS$_OK_DUP)
                printf("[Record added successfully.]\n");
        else
                printf("RMSEXP - Existing employee with same SSN, not added.\n");
}
```

Example 8-5: Utility Function: Adding Records

Example 8-6 shows the function that deletes records. This function is called when 'd' or 'D' is entered in response to the menu.

❶ A **do** loop prompts the user to type a social security number at the terminal and places the response in the response buffer.

❷ The social security number is padded with blanks.

❸ Some members in the record access block must be initialized before the program can locate the record. Here, the key of reference (0 specifies the primary key), the location and size of the search string (this is the address of the response buffer and its size), and the type of record access (in this case, keyed access) are given.

❹ The RMS **sys$find** function locates the record specified by the social security number entered from the terminal.

❺ The program checks the rms_status variable for the values RMS$_NORMAL and RMS$_RNF (record not found). A message is displayed if the record cannot be found. Any other error is a fatal error.

❻ The RMS **sys$delete** function deletes the record. The return status is checked only for success.

```
delete_employee()
{
        int i;
❶       do
                {
                printf("(DELETE) Enter Social Security Number    ");
                gets(response);
                i = strlen(response);
                } while(i == 0);

❷       while(i < SIZE_SSN)
                response[i++] = ' ';
```

```
❸       rab.rab$b_krf = 0;
        rab.rab$l_kbf = &response;
        rab.rab$b_ksz = SIZE_SSN;
        rab.rab$b_rac = RAB$C_KEY;

❹       rms_status = sys$find(&rab);

❺       if (rms_status != RMS$_NORMAL && rms_status != RMS$_RNF)
                error_exit("$FIND");
        else if (rms_status == RMS$_RNF)
                printf("RMSEXP - specified employee does not exist.\n");
        else
                {
❻               rms_status = sys$delete(&rab);
                if (rms_status != RMS$_NORMAL)
                        error_exit("$DELETE");
                }
}
```

Example 8-6: Utility Function: Deleting Records

The type_employees function in Example 8-7 displays the employee file at the terminal. This function is called from the main function when 't' or 'T' is entered in response to the menu.

❶ A running total of the number of records in the file is kept in the number_employees variable.

❷ The key of reference is changed to the alternate key so that the employees are displayed in alphabetical order by last name.

❸ The file is positioned to the beginning of the first record according to the new key of reference, and the return status of the **rms$rewind** function is checked for success.

❹ A heading is displayed.

❺ Sequential record access is specified, and the location and size of the record is given.

❻ A **for** loop controls the following operations:

- Incrementing the number_employees counter.
- Locating a record and placing it in the record structure, using the RMS **sys$get** function.
- Checking the return status of the RMS **sys$get** function.
- Displaying the record at the terminal.

❼ This **if** statement checks for records in the file. The result is a display of the number of records or a message indicating that the file is empty.

```
type_employees()
{
❶           int number_employees;

❷           rab.rab$b_krf = 1;

❸           rms_status = sys$rewind(&rab);
            if (rms_status != RMS$_NORMAL)
                    error_exit("$REWIND");
```

```
❹       printf("\n\nEmployees (Sorted by Last Name)\n\n");
        printf("Last Name      First Name          SSN                 Comments\n");
        printf("---------      ----------          ---------           --------\n\n");

❺       rab.rab$b_rac = RAB$C_SEQ;
        rab.rab$l_ubf = &record;
        rab.rab$w_usz = RECORD_SIZE;

❻       for(number_employees = 0;;number_employees++)
                {
                rms_status = sys$get(&rab);
                if (rms_status != RMS$_NORMAL && rms_status != RMS$_EOF)
                        error_exit("$GET");
                else if (rms_status == RMS$_EOF)
                        break;

                printf("%.*s%.*s%.*s%.*s\n",SIZE_LNAME,record.last_name,
                        SIZE_FNAME,record.first_name,
                        SIZE_SSN,record.ssn,
                        SIZE_COMMENTS,record.comments);
                }

❼       if (number_employees)
                printf("\nTotal number of employees = %d.\n",number_employees);
        else
                printf("[Data file is empty.]\n");
}
```

Example 8-7: Utility Function: Typing the File

Example 8-8 shows the function that prints the file on the printer. This function is called by main when 'p' or 'P' is entered in response to the menu.

❶ This function creates a sequential file with carriage-return-control, variable-length records. It spools the file to the printer when the file is closed. The file is created using the UNIX I/O **creat** function, thus associating the file with an integer file descriptor (filedes).

❷ The file descriptor is associated with a file pointer (fp), so that the file can be processed with standard I/O functions.

❸ The key of reference for the indexed file is the primary key.

❹ The **sys$rewind** function positions the file at the first record. The return status is checked for success.

❺ A heading is written to the sequential file using the standard I/O function **fprintf.**

❻ The record access, user buffer address, and user buffer size members of the record access block are initialized for keyed access to the record located in the record structure.

❼ A **for** loop controls the following operations:

- Initializing the running total and then incrementing the total at each iteration of the loop.
- Locating the records and placing them in the record structure with the RMS **sys$get** function, one record at a time.
- Checking the rms_status information for success and end-of-file.
- Writing the record to the sequential file.

❽ The number_employees counter is checked. If it is zero, a message is printed indicating that the file is empty. If it is not zero, the total is printed at the bottom of the listing.

❾ The sequential file is closed. Since it has the spl record attribute, the file is automatically spooled to the printer. The function displays a message at the terminal stating that the file was successfully spooled.

```
print_employees()
{
        int number_employees;
        int filedes;
        FILE *fp;

❶       filedes = creat("personnel.lis",0,"rat=cr","rfm=var","fop=spl");
        if (filedes == -1)
                perror("RMSEXP-failed opening listing file (creat())");
                exit(SS$_NORMAL);

❷       fp = fdopen(filedes,"w");
        if (!fp)
                perror("RMSEXP-failed opening listing file (fdopen())");
                exit(SS$_NORMAL);

❸       rab.rab$b_krf = 0;

❹       rms_status = sys$rewind(&rab);
        if (rms_status != RMS$_NORMAL)
                error_exit("$REWIND");

❺       fprintf(fp,"\n\nEmployees (Sorted by SSN)\n\n");
        fprintf(fp,"Last Name      First Name      SSN         Comments\n");
        fprintf(fp,"---------      ----------      --------    --------\n\n");
```

Example 8–8: Utility Function: Printing the File

```
❻       rab.rab$b_rac = RAB$C_SEQ;
        rab.rab$l_ubf = &record;
        rab.rab$w_usz = RECORD_SIZE;

❼       for(number_employees = 0;;number_employees++)
                {
                rms_status = sys$get(&rab);
                if (rms_status != RMS$_NORMAL && rms_status != RMS$_EOF)
                        error_exit("$GET");
                else if (rms_status == RMS$_EOF)
                        break;

                fprintf(fp,"%.*s%.*s%.*s%.*s\n",SIZE_LNAME, record.last_name,
                        SIZE_FNAME,record.first_name,
                        SIZE_SSN,record.ssn,
                        SIZE_COMMENTS,record.comments);
                }

❽       if (number_employees)
                fprintf(fp,"\nTotal number of employees = %d.\n",number_employees);
        else
                fprintf(fp,"[Data file is empty.]\n");

❾       fclose(fp);
        printf("[Listing file \"personnel.lis\" spooled to SYS$PRINT.]\n");
}
```

Example 8–8: (Cont.) Utility Function: Printing the File

Example 8-9 shows the function that updates the file. This function is called by main when 'u' or 'U' is entered in response to the menu.

❶ A **do** loop prompts for the social security number and places the response in the response buffer.

❷ The response is padded with blanks so that it will correspond to the field in the file.

❸ Some of the members in the record access block are initialized for the operation. The primary key is specified as the key of reference, the location and size of the key value are given, keyed access is specified, and the location and size of the record are given.

❹ The RMS **sys$get** function locates the record and places it in the record structure. The function checks the rms_status value for RMS$_NORMAL and RMS$_RNF (record not found). If the record is not found, a message is displayed. If the record is found, the program prints instructions for updating the record.

❺ For each field (except the social security number, which cannot be changed), the program displays the current value for that field. If the user types RET, the record is placed in the record structure unchanged. If the user makes a change to the record, the new information is placed in the record structure.

❻ The fields in the record are padded with blanks.

❼ The RMS **sys$update** function rewrites the record. The program then checks that the update operation was successful. Any error causes the program to call the fatal error-handling routine.

```
update_employee()
{
        int i;
❶       do
                {
                printf("(UPDATE) Enter Social Security Number      ");
                gets(response);
                i = strlen(response);
                } while(i == 0);

❷       while(i < SIZE_SSN)
                response[i++] = ' ';

❸       rab.rab$b_krf = 0;
        rab.rab$l_kbf = &response;
        rab.rab$b_ksz = SIZE_SSN;
        rab.rab$b_rac = RAB$C_KEY;
        rab.rab$l_ubf = &record;
        rab.rab$w_usz = RECORD_SIZE;

❹       rms_status = sys$get(&rab);

        if (rms_status != RMS$_NORMAL && rms_status != RMS$_RNF)
                error_exit("$GET");
        else if (rms_status == RMS$_RNF)
                printf("RMSEXP-specified employee does not exist.\n");
```

```
❺       else
                {
                printf("Enter the new data or <return> to leave data
                 unmodified.\n\n");
                printf("Last Name:     %.*s    ",SIZE_LNAME,record.last_name);
                gets(response);
                if (strlen(response))
                        strncpy(record.last_name,response,SIZE_LNAME);

                printf("First Name:    %.*s    ",SIZE_FNAME,record.first_name);
                gets(response);
                if (strlen(response))
                        strncpy(record.first_name,response,SIZE_FNAME);

                printf("Comments:      %.*s    ",SIZE_COMMENTS,record.comments);
                gets(response);
                if (strlen(response))
                        strncpy(record.comments,response,SIZE_COMMENTS);

❻               pad_record();

❼               rms_status = sys$update(&rab);
                if (rms_status != RMS$_NORMAL)
                        error_exit("$UPDATE");

                printf("[Record has been sucessfully updated.]\n");
                }
}
```

Example 8-9: Utility Function: Updating the File

Chapter 9

Mixed-Language Programming

Mixed-language programming is possible with VAX-11 C because the architecture of VAX-11 computers defines a set of conventions — the VAX-11 Calling Standard — that enables argument passing among procedures. With the calling standard you can write functions in C that invoke procedures written in other VAX-11 native-mode programming languages.

The VAX-11 Calling Standard defines the way a reference to a non-C function must be written in a C program. For a C function to call a non-C function that expects to receive immediate values in its argument list, the calling method is quite similar to that for calling an external C function. If the non-C function expects to receive arguments by reference or by descriptor, a function reference in the C program can still be written using familiar C operations and concepts. For example, if you want to call a C function from some other language such as PL/I, the only specific knowledge you need about C is the precise manner in which C's data types are represented on a VAX-11 computer (see Chapter 11); you can find the remaining information in the documentation for the other language, in this case the *VAX-11 PL/I User's Guide*.

The calling standard allows all communication among the native-mode VAX languages to be done within the languages themselves. The source modules for a program can be written in any of the languages, and as long as each module follows the calling standard, the linker will take the compiled object modules and construct an executable program image. It is not necessary to construct your own call interfaces to VAX/VMS system services, since the necessary definition text is provided with VAX-11 C **#include** modules contained in the library SYS$LIBRARY:CSYSDEF.TLB.

This chapter reviews the implementation of function calls in VAX-11 C and the VAX-11 Calling Standard. (If necessary, you can consult Appendix C of the *VAX-11 Architecture Handbook* for more details.) This chapter also explains the methods for calling non-C functions in VAX-11 C. It assumes that you know the C conventions and rules for passing arguments to external procedures, as described in Chapter 4. If you are interested in calling C functions from some other language, see Chapter 10.

Most of the examples in this chapter show calls to VAX/VMS system service procedures. The system services are available to all VAX/VMS installations and use all forms of argument passing. However, the examples do not fully describe the procedures themselves. For further details on system services, see the *VAX/VMS System Services Reference Manual.*

9.1 The Call Stack

The calling standard defines a call stack as a temporary storage area for each user process. The VAX-11 hardware maintains information on the call stack about each block activation in the current image.

9.1.1 Call Frames

Whenever a function is activated in a C program, the hardware creates a structure — the call frame — on the call stack for the function. The call frame for each activation contains:

- A pointer to the call frame of the previous function activation. This pointer is called the Frame Pointer (FP).
- The saved Argument Pointer (AP) of the previous activation.
- The address in storage of the point of invocation of the function, that is, the address of the next instruction following the function reference that activated the current function. This address is called the Program Counter (PC), or saved PC.
- The saved contents of some of the general registers and other control information (such as the condition codes in the processor status word, or PSW). Based on a mask specified in the control information, the system restores these registers when control returns to the caller.

Figure 9–1 illustrates the call stack and several call frames. Function A calls function B, which calls function C. When a function reaches a **return** statement or when control reaches the end of the function, the system uses the frame pointer in the call frame of the current function to locate the frame of the previous function. It then removes the call frame of the current function from the stack.

9.1.2 The Argument List

All function parameters are passed by means of an argument list, which consists of a series of up to 255 longwords. The argument list for a function activation is pointed to by a register called the Argument Pointer (AP).

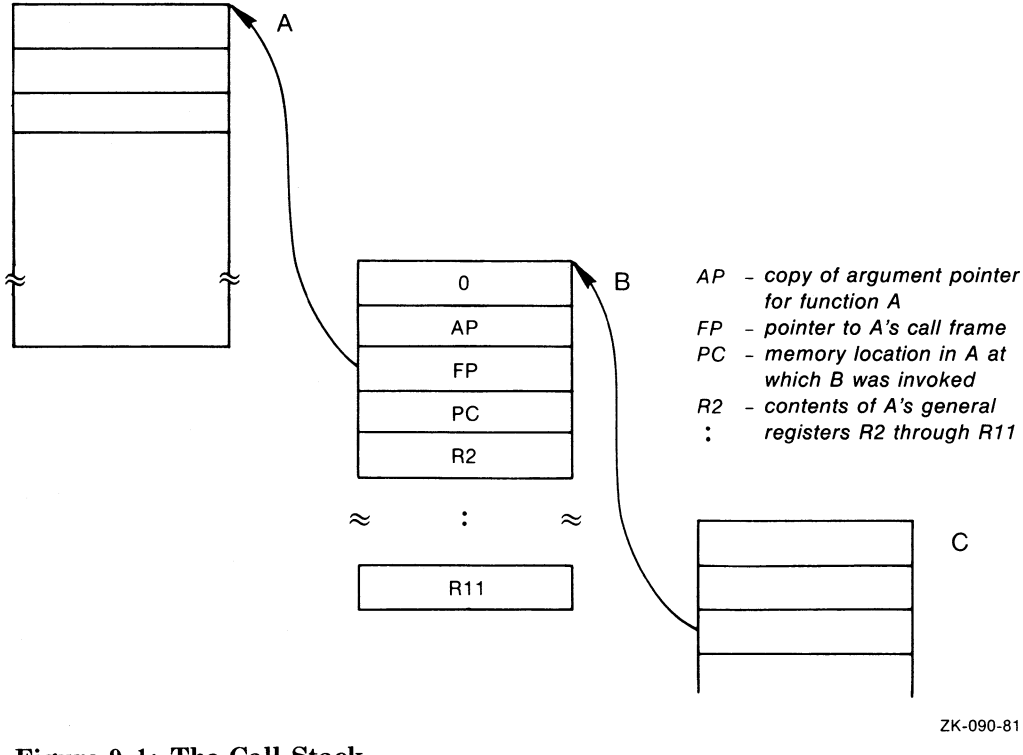


Figure 9–1: The Call Stack

The first longword in the argument list always contains, in its low-order byte, the number of arguments (longwords) that were passed; the first longword itself is not included in this number. Figure 9–2 illustrates the format of an argument list.

The calling standard defines three ways that data can be passed in an argument list. When you code a reference to a non-C procedure, you must know how each argument should be passed and write the function reference accordingly.

The three argument-passing mechanisms are:

- By immediate value. When an argument is passed by immediate value, the actual value of the argument is present in the argument list. This is the default argument-passing mechanism for all function references written in VAX–11 C.
- By reference. When an argument is passed by reference, the address of the argument is present in the argument list. The C ampersand operator (&) is used to pass the address in the argument list.
- By descriptor. When an argument is passed by descriptor, the address of a data structure describing the argument is present in the argument list. From a C program, you pass a descriptor first by creating a structure (**struct**) that meets the descriptor requirements of the called procedure and then by passing the structure's address with the ampersand operator (&).

9.2 Passing Arguments by Immediate Value

By default, all values or expressions in a VAX–11 C function's argument list are passed by immediate value. That is, the expressions are evaluated and the results placed directly in the argument list of the CALL machine instruction.

The following statement declares the entry point of the Set Event Flag SYS$SETEF[1] system service, which is used to set a specific event flag to 1. The Set Event Flag system service call requires one argument — the number of the event flag to be set — to be passed by immediate value.

```
int SYS$SETEF();
/* FUNCTION RETURNING INT */
```

1. VAX–11 C converts linker-resolved variable names (such as the entry-point names of system service calls) to uppercase. You do not have to declare them in uppercase in your program. However, linker-resolved variable names must be declared with identical cases. The documentation uses uppercase as a convention for referring to system service calls to highlight them in the text and examples.

Like all system services, SYS$SETEF returns an integer value (the return status of the service) in register 0.[1] In the declaration of external functions, the C syntax does not indicate the number or types of the arguments, nor does C compare the types of arguments with the types that the system service requires. It is your responsibility to ensure that the argument list of an external function reference contains valid arguments.

In the *VAX/VMS System Services Reference Manual* you can find the specification of each service's arguments. SYS$SETEF, for example, takes one argument, an event flag number. It returns one of four status values, which are represented by the following symbolic constants:

Returned	Status	Description
SS$_WASCLR	Success	Flag was previously clear
SS$_WASSET	Success	Flag was previously set
SS$_ILLEFC	Failure	Illegal event flag number
SS$_UNASEFC	Failure	Event flag not in associated cluster

The system services manual also defines event flags as integers in the range 0 to 127, grouped in clusters of 32. Clusters 0 and 1, comprising flags 0 to 31 and 32 to 63, respectively, are local clusters available to any process, with the restriction that flags 24 to 31 are reserved for use by VAX/VMS. There are many ways of passing valid event flag numbers from your C program to SYS$SETEF. One way is to use **enum** to define a subset of integers:

```
enum cluster0 {completion,breakdown,beginning} event;
```

Once the flag numbers have been defined, the SYS$SETEF service can be called by writing:

```
.
.
int status;
event = completion;
.
.
status = SYS$SETEF(event);      /* SET EVENT FLAG */
.
.
```

1. Most system services return an integer completion status; therefore, the system service does not always have to be declared before it is used. The examples in this chapter declare system services for completeness.

Figure 9–3 shows an argument being passed by immediate value — in this case, the event flag number passed to SYS$SETEF.

9.2.1 Checking System Service Return Values

The custom in VAX/VMS programming is to compare the return status of a system service with a global symbol, not with the literal value associated with a particular return status. Consequently, a high-level language program should define the possible return status values for a service as symbolic constants. In VAX–11 C, you can do so by including the text library module ssdef; Example 9–1 shows how this is done.

```
/* DEFINE SYSTEM SERVICE STATUS VALUES */
#include ssdef
#include stdio

/* DECLARATION OF THE SERVICE (not required) */
int SYS$SETEF();

/* CALLING FUNCTION */
main()
{
     /* STATUS OF $SETEF */
     int efstatus;

     /* ARGUMENT VALUES FOR $SETEF */
     enum cluster0 {completion,breakdown,beginning}
     event;
     .
     .
     event = completion;
```

Example 9–1: Checking System Service Return Values

```
/* SET EVENT FLAG */
efstatus = SYS$SETEF(event);

/* TEST RETURN STATUS */
if(efstatus) ==. SS$_WASSET)
     {
     fprintf (stderr,"Flag was already set\n");
     }
else if(efstatus == SS$_WASCLR)
     {
     fprintf(stderr,"Flag was previously clear\n");
     }
else fprintf(stderr,
       "Could not set completion event flag.\n \
        Possible programming error.\n");
     exit(efstatus);
```

Example 9-1: (Cont.) Checking System Service Return Values

The system service return status values (SS$_WASSET and SS$_WASCLR) in Example 9-1 are defined by the **#include** text module ssdef.

In the example, the statement executed when an error occurs also shows behavior typical of programs running under VAX/VMS. With the statements

```
else fprintf(stderr,
       "Could not set completion event flag.\n \
        Possible programming error.\n");
     exit(efstatus);
```

the example program attempts to provide a program-specific error message and then passes the offending error status to the caller. If the program were to be executed by the DCL, then any status value returned by the program would be interpreted by DCL. DCL prints a standard error message on the terminal to provide you with more information about the reason for the failure. For example, if the program were to encounter the SS$_ILLEFC return status, the following messages would be displayed:

```
Could not set 'completion' event flag.
Possible programming error.
%SYSTEM-F-ILLEFC, illegal event flag cluster.
```

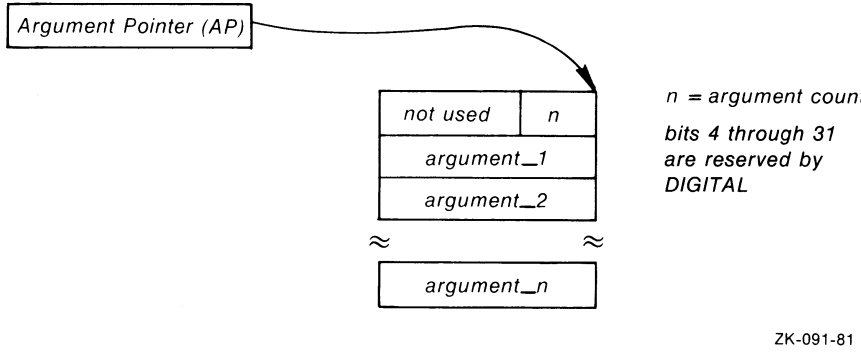


Figure 9–2: An Argument List

```
main( )
{
  .
  .
  SYS$SETEF (4) ;
}
```

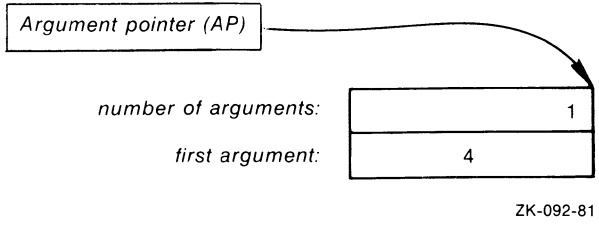


Figure 9–3: Passing Arguments by Immediate Value

9.2.2 Passing Floating-Point Arguments by Immediate Value

Because argument lists consist of longwords, the calling standard dictates that immediate-value arguments be expressible in 32 bits. A single-precision floating-point (F-floating) value is only 32 bits long, but all arguments of type **float** are promoted by C to **double** (on a VAX-11, 64 bits). This double-precision value is passed as two immediate values (two longwords).

NOTE

The passing of double-precision immediate values is a violation of the usual VAX-11 procedure-calling standard, but is an allowed exception for VAX-11 C.

On rare occasions, the **float**-to-**double** promotion requires some additional programming. For instance, the function OTS$POWRJ, in the VAX-11 Common Run-Time Procedure Library, computes the value of a floating-point number raised to the power of a signed longword (in C terms, a **float** to the power of an **int**). This function (and others like it) is called implicitly by high-level VAX languages that have an exponentiation operator as part of the language. It requires that both its arguments be passed as immediate values, and it returns a single-precision (**float**) result. To pass a floating-point base to the procedure, you must use some method that avoids the promotion of **float** arguments. One such method is to use a structure, as shown in Example 9-2:

By default, structures, like everything else, are passed by immediate value. Thus, in Example 9-2 the argument is not interpreted as a **float** and is not promoted to **double**.

The great majority of run-time functions that operate on floating-point values take their arguments by reference, so the procedure illustrated by Example 9-2 is not usually necessary. You should note, in addition, that the example does not illustrate the methods for handling arithmetic errors that result from the operation performed. For more information on error handling in this context, and on the run-time library in general, see the *VAX-11 Run-Time Library Reference Manual.*

```
#include stdio

/* FUNCTION RETURNING FLOAT; CALLING SEQUENCE IS
 * OTS$POWRJ(base,power), WHERE base
 * IS A float AND POWER IS AN int
 */
float OTS$POWRJ();

/* PROGRAM CALLING OTS$POWRJ */
main()
{
     /* OTS$POWRJ RESULT */
     float result;

     /* POWER ARGUMENT */
     int power;

     /* STRUCTURE USED TO PASS FLOAT BY VALUE */
     struct { float f; } base;

     /* ASSIGN CONSTANT (IMPLIED float) TO BASE */
     base.f = 3.145;
     power = 2;
     result = OTS$POWRJ(base,power);

     printf ("Result= %f\n",result);
}
```

Example 9–2: Passing Floating-Point Arguments by Immediate Value

9.3 Passing Arguments by Reference

Some system services and run-time library procedures expect arguments passed by reference. This means that the argument list (in the CALL machine instruction) contains the address of the argument rather than its value. This mechanism is also used by default by some programming languages, such as PL/I, and is available at the programmer's option in others, such as PASCAL.

In VAX-11 C, you can use the ampersand operator (&) to pass an argument by reference, that is, the ampersand operator causes the argument's address to be passed. Note also that an array or function name in an argument list always results in passing the address of the array or function; the ampersand is not required in such cases.

In the special case of argument lists, VAX-11 C allows the ampersand operator to be used on constants as well. (However, you should limit this use of the ampersand to calls to VAX/VMS system functions to ensure portability of your VAX-11 C programs to other C compilers.)

For example, the Read Event Flags (SYS$READEF) system service requires that its first argument be passed by immediate value and its second argument be passed by reference. SYS$READEF returns the status of all the event flags in a particular cluster. (Event flags are numbered from 0 to 127 and arranged in clusters of 32, such that flags 0 to 31 comprise cluster 0, flags 32 to 63, cluster 1, and so forth.) The first SYS$READEF argument is any event flag number in the cluster of interest. The second argument is the address of a longword that receives the status of all 32 event flags in that cluster. In addition to the event-flag status value, the system service returns one of the following status values, expressed here as global symbols:

Returned	Status	Description
SS$_WASCLR	Success	Specified event flag was clear
SS$_WASSET	Success	Specified event flag was set
SS$_ACCVIO	Failure	Could not write to status longword
SS$_ILLEFC	Failure	Event flag number was illegal
SS$_UNASEFC	Failure	Cluster of interest not accessible

Example 9-3 shows a call to the SYS$READEF system service from a C program.

```
/* DEFINE SYSTEM SERVICE STATUS VALUES */
#include ssdef
#include stdio

/* DECLARATION OF $READEF (not required) */
int SYS$READEF();

/* CALLING FUNCTION */
main()
{
     /* LONGWORD THAT RECEIVES THE
        STATUS OF THE EVENT FLAG CLUSTER */
     unsigned cluster_status;

     /* RETURN STATUS OF $READEF */
     int return_status;

     /* ARGUMENT VALUES FOR $READEF */
     enum cluster0 {completion,breakdown,beginning}
     event;
     .
     .
     /* EVENT FLAG IN CLUSTER 0 */
     event = completion;

     /* OBTAIN STATUS OF CLUSTER 0:
      * PASS VALUE OF event AND
      * ADDRESS OF cluster_status
      */
     return_status = SYS$READEF(event,
                                &cluster_status);

     /* CHECK FOR SUCCESSFUL CALL */
     if(return_status != SS$WASCLR &&
        return_status != SS$WASSET)
     {
     /* ERROR PROCESSING */
       .
       .
     }
     else
     {
     /* CHECK BITS OF INTEREST IN cluster_status */
      .
      .
     }
}
```

Example 9–3: Passing Arguments by Reference

Figure 9–4 illustrates argument passing by reference — in this case, to the SYS$READEF system service.

9.4 Passing Arguments by Descriptor

A descriptor is a structure that describes the data type, size, and address of a datum. According to the VAX-11 Calling Standard, you must pass a descriptor by placing its address in the argument list. To pass an argument by descriptor from a VAX-11 C program, you perform the following steps:

1. Write a **struct** declaration that models the required descriptor. This involves including the text library module descrip to define **struct** tags for all the forms of descriptors.
2. Assign appropriate values to the structure members.
3. Use the structure name, with an ampersand operator (&) in the function reference, to put the structure's address in the argument list.

In default cases, VAX-11 C never passes arguments by descriptor. For example, when structure or union names are written in a function's argument list without the ampersand operator, the structure or union is passed by immediate value to the called function. You pass arguments by descriptor only when the called function is written in another language and explicitly requires this mechanism.

There are several classes of descriptor. Each class requires that certain bits be set in the first longword of the descriptor. These classes, and the format of the descriptor defined by each, are described in the *VAX-11 Architecture Handbook*. In accordance with the information in the handbook, descriptors can be modeled as follows in VAX-11 C:

```
struct dsc$descriptor
    {
    unsigned short dsc$w_length;    /*LENGTH OF DATUM*/
    char dsc$b_dtype                /*DATA TYPE CODE*/
    char dsc$b_class                /*DESCRIPTOR CLASS
                                      CODE*/
    char *dsc$a_pointer             /*HAS ADDRESS OF
                                      FIRST BYTE*/
    };
```

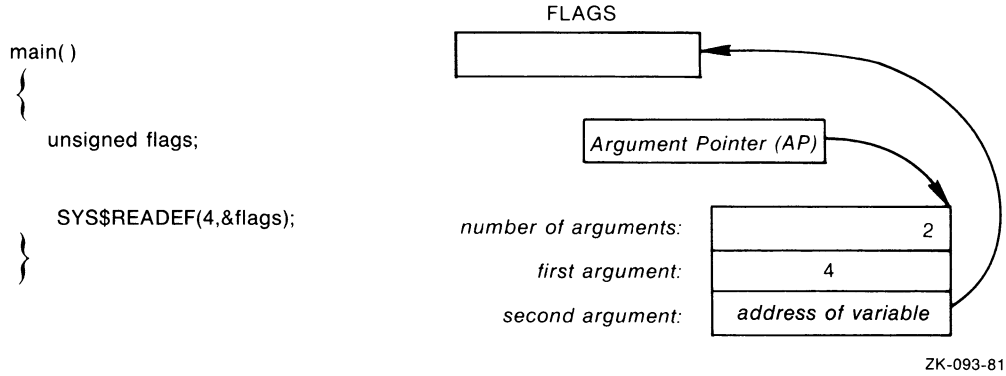


Figure 9-4: Passing Arguments by Reference

In this model, dsc$w_length is a 16-bit word containing the length of the entire datum; the unit (for example, bit or byte) in which the length is measured depends on the descriptor class. The dsc$b_dtype member is an 8-bit byte containing a numeric code; the code denotes the data type of the datum. The class member dsc$b_class is another byte code giving the descriptor class. The valid class codes are as follows:

Class Code	Symbolic Name	Descriptor Class
1	DSC$K_CLASS_S	Scalar, string
2	DSC$K_CLASS_D	Dynamic string descriptor
3	—	Reserved by DIGITAL
4	DSC$K_CLASS_A	Array
5	DSC$K_CLASS_P	Procedure
6	DSC$K_CLASS_PI	Procedure incarnation
7	DSC$K_CLASS_J	Label
8	DSK$K_CLASS_JI	Label incarnation
9–191	—	Reserved by DIGITAL
192–255	—	Reserved for customer applications

The last member of the structure model, dsc$a_pointer, points to the first byte of the datum.

To pass an argument by descriptor, you define and assign values to the datum following the normal C programming practices. You must define a structure of the form shown above and assign the datum's address to the pointer member. You must also assign appropriate values to the other members, dscw_length, dscb_dtype, and dsc$b_class. See the *Architecture Handbook* for the specific requirements of each descriptor class.

For example, the Set Process Name (SYS$SETPRN) system service, which enables a process to establish or change its process name, accepts a process name as a fixed-length character string passed by descriptor. The character string can have from 1 to 15 characters. The system service returns the status values denoted by the global names SS$_NORMAL, SS$_ACCVIO, SS$_DUPLNAM, and SS$_IVLOGNAM (for normal completion, inaccessible descriptor, duplicate process name, and invalid length, respectively). Example 9-4 shows a call to this system service from a C program.

```
/*DEFINE SYSTEM SERVICE STATUS VALUES*/
#include ssdef
/*DEFINE STRUCTURES FOR DESCRIPTORS*/
#include descrip
#include stdio

/*DECLARATION OF THE SERVICE (not required)*/
int SYS$SETPRN();

/*PROGRAM CALLING $SETPRN*/
main()
{
     /*RETURN STATUS OF $SETPRN*/
     int ret;

     /*NAME DESCRIPTOR*/
     struct dsc$descriptor_s name_desc;

     /*NEW PROCESS NAME*/
     char *name = "NEWPROC";
     .
     .
     /*LENGTH OF NAME WITHOUT NUL TERMINATOR*/
     name_desc.dsc$w_length = strlen(name);

     /*PUT ADDRESS OF SHORTENED
       STRING IN DESCRIPTOR*/
     name_desc.dsc$a_pointer = name;

     /*STRING DESCRIPTOR CLASS*/
     name_desc.dsc$b_class = DSC$K_CLASS_S;

     /*DATA TYPE IS ASCII STRING*/
     name_desc.dsc$b_dtype = DSC$K_DTYPE_T;
     .
     .
     ret = SYS$SETPRN(&name_desc);

     /*TEST RETURN STATUS*/
     if(ret != SS$_NORMAL)
     fprintf(stderr, "Failed to set process
                      name\n");
     exit(ret);
     .
     .
}
```

Example 9–4: Passing Arguments by Descriptor

Note that the call to SYS$SETPRN *must* use the ampersand operator; otherwise name_desc, rather than its address, is passed.

Although this example explicitly sets individual fields in its name_desc string descriptor, in practice, the run-time initialization of compile-time constant string descriptors is not performed in this manner. Instead, the fields of compile-time constant descriptors are usually initialized with statically initialized structures.

For the purpose of string descriptor initialization, VAX-11 C provides a simple preprocessor macro in the **#include** text library module descrip. This macro is named $DESCRIPTOR. It takes two arguments, which it uses in a standard C structure declaration. The first argument is an identifier specifying the name of the descriptor to be declared and initialized. The second argument is a pointer to the data byte to be used as the value of the descriptor. (Because a character-string constant is interpreted as an initialized pointer to **char**, you may specify the second argument as a simple string constant.) The $DESCRIPTOR macro may be used in any context where a declaration may be used. The scope of the declared string descriptor identifier name is identical to the scope of a simple **struct** definition as expanded by the macro.

Example 9–5 shows a variant of the program in Example 9–4. Here, the $DESCRIPTOR macro is used to create a compile-time string descriptor and to pass it to the SYS$SETPRN system service routine. In Example 9–5, the program simply returns the status value returned by SYS$SETPRN to DCL for interpretation.

```
/*DEFINE $DESCRIPTOR MACRO*/
#include descrip

/*DECLARE THE SERVICE (not required)*/
int SYS$SETPRN ();

/*PROGRAM CALLING $DESCRIPTOR*/
main()
{
    /*INITIALIZE STRUCT name_desc AS STRING
      DESCRIPTOR*/
    static $DESCRIPTOR(name_desc,"NEWPROC");

    return SYS$SETPRN(&name_desc);
}
```

Example 9–5: Passing Compile-Time String Descriptors

The $DESCRIPTOR macro is used in further examples in this chapter.

9.5 Variable-Length Argument Lists

Although most system services and other external procedures require a specific number of arguments, some accept a variable number of optional arguments. Because C function declarations never show the number of parameters expected by external functions, the way you call an external function from a VAX-11 C program depends on the semantics of the called function. Briefly stated, you must always supply the number of arguments that the external function expects. The rules are as follows:

- When optional arguments occur between required arguments, they cannot simply be omitted, or nulled. If omitting such an argument is necessary — for example, to select a default action — the argument must be written as a zero.
- When optional arguments occur at the end of an argument list, the format of the function reference depends on the action of the called function:
  - — If the called function checks the number of arguments passed, you can omit optional trailing arguments from the function reference. (Note that system services generally do not check the length of the argument list.)
  - — If the called function does not check the number of arguments passed, all arguments must be present in the function reference.

For example, the function STR$CONCAT, in the Common Run-Time Library, concatenates from 2 to 254 strings into a single string. Its call format is as follows (see also the *VAX-11 Run-Time Library Reference Manual*):

```
ret = STR$CONCAT(dst,src1,scr2[,src3,...src254);
```

where dst is the destination for the concatenated string, and src1, src2,...src254 are the source strings. (All arguments are passed by descriptor.) All but the first two source strings are optional. The function checks to see how many arguments are present in the call; if fewer than three (the destination and two sources) are present, the function returns an error status value. Example 9–6 shows a call to the STR$CONCAT function from VAX-11 C.

```
#include stdio
#include descrip
#include ssdef

/*DECLARATION OF STR$CONCAT (not required)*/
int STR$CONCAT();

main()
{
    /*RETURN STATUS OF STR$CONCAT*/
    int ret;

    /*DESTINATION ARRAY OF CONCATENATED STRINGS*/
    char dest[21];

    /*CREATE COMPILE-TIME DESCRIPTORS*/
    $DESCRIPTOR(dst,dest);
    static $DESCRIPTOR(src1, "abcdefghij");
    static $DESCRIPTOR(src2, "klmnopqrst");

    /*CONCATENATE STRINGS*/
    ret = STR$CONCAT(&dst,&src1,&src2);

    /*TEST RETURN STATUS VALUE*/
    if (ret != SS$_NORMAL)
        fprintf(stderr,"Failed to concatenate
                strings.\n");
        exit(ret);

    /*PROCESS STRING*/
    else
        dest[20] = '\0';
    printf("Resultant string: %s\n",dest);
}
```

Example 9–6: Use of Variable-Length Argument Lists

9.6 Return Status Values

The VAX–11 returns status values from system service procedures in general register R0. This return status value indicates the success or failure of the operation performed by the called procedure. In VAX–11 C, passing a return status value in R0 is equivalent to a function returning **int**.

To obtain a return status value from any system procedure, you can declare the procedure as a function, as shown in the following example:

```
int SYS$SETEF();
```

After declaring a procedure in this way, you can invoke the procedure as a function and obtain a return status value. (In C, such a declaration is needed only as program documentation; SYS$SETEF could simply be called without explicit declaration and would be interpreted by default as a function returning **int**.)

This section describes:

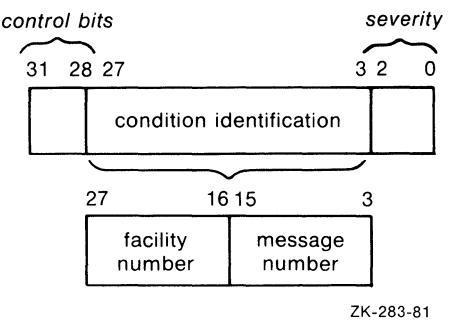
- The format of a return status value, that is, the meaning of particular bits within the value.
- The way to manipulate return status values.
- Recommended techniques for testing a return status value for success or failure or for a specific condition.

9.6.1 Format of Return Status Values

All VAX/VMS system procedures and programs use a longword value to communicate return status information. When a VAX-11 C main function executing under the control of the DCL command interpreter executes a **return** statement to return control to the command level, the command interpreter uses the return status value to conditionally display a message on the current output device.

To provide a unique means of identifying every return condition in the system, bit fields within the value are defined as follows:

These fields are:



control bits (31–28)
These define special action(s) to be taken. At present, only bit 28 is used. When set, it inhibits the printing of the message associated with the return status value at image exit. Bits 29 through 31 are reserved for future use by DIGITAL and must be zero.

facility number(27–16)
This is a unique value assigned to the system component, or facility, that is returning the status value. Within this field, bit 27 has a special significance. If bit 27 is clear, the facility is a DIGITAL facility: the remaining value in the facility number field is a number assigned by the operating system. If bit 27 is set, the number indicates a customer-defined facility.

message number (15–3)
This is an identification number that specifically describes the return status or condition. Within this field, bit 15 has a special significance. If bit 15 is set, the message number is unique to the facility that is issuing the message. If bit 15 is clear, the message is issued by more than one system facility.

severity (2–0)
This is a numeric value indicating the severity of the return status. The possible values in these three bits, and their meanings, are as follows:

Value	Meaning
0	Warning
1	Success
2	Error
3	Informational
4	Severe error
5–7	Reserved

Note that odd values indicate success (an informational condition is considered a successful status) and that even values indicate failures (a warning is considered an unsuccessful status).

The following names are associated with these fields:

control bits	CONTROL
bit 28 (inhibit message)	INHIB_MSG
facility number	FAC_NO
bit 27 (customer facility)	CUST_DEF
message number	MSG_NO
bit 15 (facility specific)	FAC_SP
severity	SEVERITY
bit 0 (success)	SUCCESS

When testing return values in a VAX-11 C program, you can either test only for successful completion of a procedure, or you can test for specific return status values.

9.6.2 Manipulating Return Status Values

It is possible to construct a structure or union that describes a return status value, but in practice this method of manipulating return status values is unwieldy. A status value is usually constructed or checked using bitwise operators. VAX-11 C provides the **#include** module stsdef, which contains preprocessor definitions to make this job easier. All of the preprocessor symbols are named according to the VAX/VMS naming convention, as follows:

STS$type_name

STS
identifies standard return status values.

type
is one of the following characters denoting the type of the constant:

K	represents a constant value
M	represents a bit mask
S	represents the size of a field
V	defines the bit offset to the field

name
is an abbreviation for the field name.

For example, the following constants are defined in stsdef for the facility number field, FAC_NO, which spans bits 16 through 27:

```
#define STS$S_FAC_NO   12          /*SIZE OF THE FIELD
                                     IN BITS*/

#define STS$V_FAC_NO   16          /*BIT OFFSET TO THE
                                     BEGINNING OF THE
                                     FIELD*/

#define STS$M_FAC_NO   0xFFF0000   /*BIT MASK OF THE
                                     FIELD*/
```

Figure 9–5 shows how the status value would be represented internally.

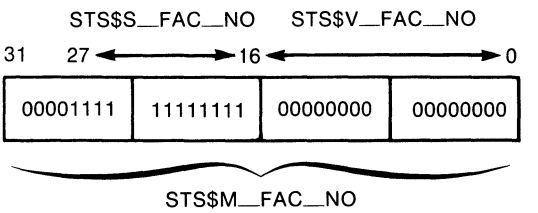


Figure 9–5: Internal Representation of a Status Value

The following expression can be used to extract the facility number from a particular status value contained in the variable named status:

```
(status & STS$M_FAC_NO) >> STS$V_FAC_NO
```

Note that the parentheses are required for the expression to be evaluated properly; the relative precedence of the bitwise AND operator (&) is lower than the precedence of the binary shift operator (>>).

9.6.3 Testing for Success or Failure

To test a return status value for success or failure, you need only test the SUCCESS bit. A value of true in this bit indicates that the return value is a successful value.

Example 9–7 shows a program that checks the SUCCESS bit.

```
#include stdio
#include descrip
#include stsdef

main()
{
     int status;
     $DESCRIPTOR(name,"student");

     status = SYS$SETPRN(&name);

     if (status & STS$M_SUCCESS)
        {
        /* SUCCESS CODE */
        fprintf(stderr,"Successful completion");
        }

     else
        /* FAILURE CODE */
        fprintf(stderr,"Failed to set process
                        name.\n"),
        exit(status);
}
```

Example 9–7: Testing for Success

The failure code in Example 9–7 causes the printing of a program-specific message that indicates the condition that caused the program to terminate. The error status is passed to the DCL (via the **exit** function), which then interprets the status value.

9.6.4 Testing for Specific Return Status Values

Each numeric return status value defined by the system has a symbolic name associated with it. The names of these values are defined as system global symbols, and you can access their values by referring to their symbolic names.

The global symbol names for VAX/VMS return status values have the format:

```
facility$__code
```

facility
The facility is an abbreviation or acronym for the system facility that defined the global symbol.

code
The code is a mnemonic for the specific status value.

Some examples of facility codes used in global symbol names are:

Facility Code	Used By
SS	System services; these status codes are listed in the *VAX/VMS System Services Reference Manual.*
RMS	File system procedures; these status codes are listed in the *VAX-11 Record Management Services Reference Manual.*
SOR	SORT procedures; these status codes are listed in the *VAX-11 SORT User's Guide.*

The definitions of the global symbol names for the facilities listed above are located in the default system object module libraries, and thus are automatically located when you link a VAX-11 C program that references them.

When you write a VAX-11 C program that calls system procedures and you want to test for specific return status values using the symbol names, you must:

1. Determine, from the documentation of the procedure, the status values that can be returned, and choose the values for which you want to provide specific tests.

2. Declare the symbolic name for each value of interest. The ssdef and rmsdef **#include** modules define, respectively, the system service and RMS return status values. (The return status values in these two modules are defined with the **#define** control line.) If you are checking return status values from other facilities, such as the SORT utility, you must explicitly declare the return values as **globalvalue**s. For example:

```
globalvalue int SOR$_OPENIN;
```

3. Reference the symbols in your program.

Example 9–8 shows a program that checks for specific return status values (defined by the ssdef **#include** module).

```
#include ssdef
#include stdio
#include descrip

$DESCRIPTOR(message,"\07<<Lunch-time>>\07");

main()
{

    int status = sys$brdcst(&message,0);

    if (status != SS$_NORMAL)
        {
        if (status == SS$_NOPRIV)
            fprintf(stderr,
                "Can't broadcast; requires OPER
                 privilege,");
        else
            fprintf(stderr,
                "Can't broadcast; some fatal
                 error,");

        exit(status);
        }
}
```

Example 9–8: Testing for Specific Return Status Values

Chapter 10

Storage Allocation

This chapter provides general information on the use of program sections by the VAX-11 C compiler and the VAX-11 Linker. Examples are shown in which VAX-11 C shares program sections with VAX-11 FORTRAN, VAX-11 PL/I, and VAX-11 MACRO. For program sections to be shared among languages, you need to know the way each non-C language stores various data types, and the way each language allocates program sections for external data. For full details on sharing program sections with other languages, see the documentation supplied with those languages (for example, the *VAX-11 PL/I Encyclopedic Reference* or the *VAX-11 FORTRAN User's Guide*).

10.1 Program Sections

When the VAX-11 C compiler creates an object module, it groups data in the object module into contiguous areas called program sections, or psects. The grouping depends on the attributes of the data and on whether the psects contain executable code or read/write variables.

The compiler also writes into each object module information about the program sections contained in it. The linker uses this information when it binds object modules into an executable image. As the linker allocates virtual memory for the image, it groups together program sections that have similar attributes.

Finally, the compiler adds any global names to the object module's table of global symbols. Names declared with **globalref**, **globaldef**, and **globalvalue** are written in this table; others are not. Note that **globalvalue** adds a name to the global symbol table but does not allocate storage in any program section; if an initializer appears with **globalvalue**, the name added to the symbol table is a global symbol for the given value; if no initializer appears, the name is a global name for a value defined elsewhere in the system. (For more information on global symbols, see Chapter 11.)

10.1.1 Attributes of Program Sections

Table 10–1 lists the attributes that can be applied to program sections.

Table 10–1: Program Section Attributes

Attribute	Meaning
PIC or NOPIC	The program section or the data to which it refers does not depend on any specific virtual memory location (PIC), or else the program section depends on one or more virtual memory locations (NOPIC).[1]
CON or OVR	The program section will be concatenated with other program sections with the same name (CON) or will be overlaid on the same memory locations (OVR).
REL or ABS	The data in the program section can be relocated within virtual memory (REL) or are not considered in the allocation of virtual memory (ABS).
GBL or LCL	The program section is part of one cluster, is referenced by the same program section name in different clusters (GBL), or is local to each cluster in which its name appears (LCL).
EXE or NOEXE	The program section contains executable code (EXE) or does not contain executable code (NOEXE).
WRT or NOWRT	The program section contains data that can be modified (WRT) or data that cannot be modified (NOWRT).
RD or NORD	These attributes are not currently used.
SHR or NOSHR	The program section can be shared in memory (SHR) or cannot be shared in memory (NOSHR).
USR or LIB	These attributes are reserved for future use.
VEC or NOVEC	The program section contains privileged change mode vectors (VEC) or does not contain those vectors (NOVEC).

1. C programs can be bound into PIC or NOPIC shareable images. NOPIC occurs if declarations such as the following are used:

```
char *x = &y;
```

This statement relies on the address of y to determine the value of the pointer x.

10.1.2 Program Sections Created by VAX-11 C

VAX-11 C always creates the following program sections:

- $CODE — contains all executable code and constant data (including variables defined with the **readonly** keyword).
- $DATA — contains all static variables, as well as global variables defined without the **readonly** keyword.
- $CHAR_STRING_CONSTANTS — contains C character-string constants written in the program, such as:

  ```
  "This is a string."
  ```

  This program section has the same attributes as $DATA (see Table 10–2).

VAX-11 C also creates additional program sections for **extern** variables and global variables (when the global variables' declarations specify a program section name explicitly). Table 10–2 summarizes the differences in program section attributes that correspond to differences in VAX-11 C storage classes. All program sections created by VAX-11 C have the attributes PIC, REL, RD, USR, and NOVEC.

All program sections generated by VAX-11 C (except $CODE) are aligned on longword boundaries; the $CODE psect is aligned on byte boundaries.

Table 10–2: Program Sections for VAX-11 C Variables

Storage Class Keywords	Program Section Name	Program Attributes
[**extern**][1]	*name*[2]	OVR, GBL, SHR, NOEXE, WRT
[**extern**][1] **readonly**	*name*[2]	OVR, GBL, SHR, NOEXE, NOWRT
static	$DATA	CON, LCL, NOSHR, NOEXE, WRT
static readonly	$CODE	CON, LCL, SHR, EXE, NOWRT
globaldef	$DATA	CON, LCL, NOSHR, NOEXE, WRT
globaldef {"name"}	*name*[2]	CON, GBL, SHR, NOEXE, WRT
globaldef readonly	$CODE [or *name*][2]	CON, LCL, SHR, EXE, NOWRT

1. If **extern** is present, the declaration is a reference to a previously existing datum in a program section with these attributes; if **extern** is absent, storage is allocated in such a program section.
2. *name* is either the identifier of the variable declared with the specified keyword(s) or the name specified in **globaldef** {"name"}.

10.1.3 Link-Time Scope of Names

The term *link-time scope* is sometimes used to describe whether a particular object is accessible by more than one module in a program image. For simple variables and arrays, the identifier is recognized by the linker. For structures and unions, only the identifier, *not* the tag or the members, is recognized by the linker. Therefore, only the names of objects have a link-time scope.

The link-time scope of a variable depends on its storage class, as follows:

- The scope of a **static** variable is restricted to the compilation unit in which it appears. If an identical declaration of the **static** variable appears in a different object module, a different object is defined.
- For all other storage classes covered in Table 10–2, the link-time scope is the entire image, since all modules in the program have access to the program sections in which they reside. You must still declare the name with congruent declarations in every module, but each declaration refers to the same object. For example, **extern** is used to declare the name of an object defined in a standard external data definition, where the definition can be in another module. **globalref** is used to declare the name of an object defined elsewhere with **globaldef**.

If two variables are declared with different attributes (for example, read-only in one instance, not read-only in another), the VAX-11 Linker will issue diagnostics at link-time, usually the MULPSC (conflicting psect attributes) warning diagnostic.

10.2 Sharing Program Sections with FORTRAN Common Blocks

In a FORTRAN program, separately compiled procedures can share data in declared common blocks which specify the names of one or more variables to be placed in them. Each named common block represents a separate program section. Each procedure that declares the common block with the same name can access the same variable.

As shown in Example 10–1, a VAX-11 C **extern** variable corresponds to a FORTRAN common block with the same name.

STRING.C contains:

```
main()
{
     extern char xyz[20];

     strncpy(xyz,"This is a string      ",sizeof xyz);
     prstring();
}
```

PRSTRING.FOR contains:

```
SUBROUTINE PRSTRING
CHARACTER*20 STRING
COMMON /XYZ/ STRING

   TYPE 20, STRING
20 FORMAT (' ',A20)
   RETURN
   END
```

Example 10–1: Sharing Data with a FORTRAN Program in Named Program Sections

In Example 10–1, the VAX–11 C **extern** variable xyz corresponds to the FORTRAN common block named XYZ. The FORTRAN procedure displays the data in the block.

To share data in more than one variable in a program section with a FORTRAN program, the VAX–11 C variables must be declared within a structure, as shown in Example 10–2.

NUMBERS.C contains:

```
struct xs
     {
     int first;
     int second;
     int third;
     };

main()
{
     extern struct xs numbers;

     numbers.first = 1;
     numbers.second = 2;
     numbers.third = 3;
     fnum();

}
```

FNUM.FOR contains:

```
SUBROUTINE FNUM
INTEGER*4 INUM,JNUM,KNUM
COMMON /NUMBERS/ INUM,JNUM,KNUM

   TYPE 10, (INUM,JNUM,KNUM)
10 FORMAT (3I8)
   RETURN
   END
```

Example 10–2: Sharing Data with a FORTRAN Program in a VAX-11 C Structure

In Example 10–2, the **int** variables declared in the VAX-11 C structure numbers correspond to the FORTRAN INTEGER*4 variables in the COMMON of the same name. Note that in a FORTRAN common block, all variables must be either integers or character strings. Variables of different data types cannot be grouped into the same block.

10.3 Sharing Program Sections with PL/I Externals

A VAX-11 PL/I variable with the EXTERNAL attribute corresponds to a FORTRAN common block and to a VAX-11 C **extern** variable. Examples 10-3 and 10-4 illustrate the sharing of a program section between VAX-11 C and VAX-11 PL/I.

A PL/I EXTERNAL CHARACTER attribute corresponds to a VAX-11 C **extern char** variable, but PL/I character strings are not necessarily NUL-terminated. In Example 10-3, VAX-11 C and VAX-11 PL/I use the same variable to manipulate the character string that resides in a program section named XYZ.

STRING.C contains:

```
main()
{
     extern char xyz[20];

     strncpy(xyz,"This is a string ",sizeof xyz);
     prstring();
}
```

PRSTRING.PLI contains:

```
PRSTRING: PROCEDURE;

     DECLARE XYZ EXTERNAL CHARACTER(20);

     PUT SKIP LIST(XYZ);
     RETURN;

END PRSTRING;
```

Example 10-3: Sharing Data with a PL/I Program in Named Program Sections

The PL/I procedure PRSTRING writes out the contents of the external variable XYZ.

PL/I also has a structure type similar (in its internal representation) to the **struct** in VAX-11 C. Moreover, VAX-11 PL/I can output aggregates, such as structures and arrays, in fairly simple stream-output statements; see Example 10-4.

NUMBERS.C contains:

```
main()
{
     extern struct xs numbers;

     numbers.first = 1;
     numbers.second = 2;
     numbers.third = 3;
     fnum();

}
```

FNUM.PLI contains:

```
FNUM: PROCEDURE;
    /* EXTERNAL STRUCTURE CONTAINING THREE INTEGERS */
    DECLARE 1 NUMBERS EXTERNAL,
            2 FIRST FIXED(31),
            2 SECOND FIXED(31),
            2 THIRD FIXED(31);

    PUT SKIP LIST('Contents of structure:',NUMBERS);
    RETURN;
END FNUM;
```

Example 10-4: Sharing Data with a PL/I Program in a VAX-11 C Structure

The PL/I procedure FNUM writes out the complete contents of the external structure NUMBERS; the structure members are written out in the order of their storage in memory, which is the same as for a C **struct**.

10.4 Sharing Program Sections with MACRO Programs

In a MACRO program, the .PSECT directive sets up a separate program section that can store data or MACRO instructions. The attributes in the .PSECT directive describe the contents of the program section.

You can set up a psect in a MACRO program to allow data to be shared with a VAX-11 C program, as shown in Example 10-5.

NUMBERS.C contains:

```
struct
    {
    int     first;
    int     second;
    int     third;
    } example;

main()
{
    set_value();

    printf("example.first = %d\n",example.first);
    printf("example.second = %d\n",example.second);
    printf("example.third = %d\n",example.third);
}
```

SETVALUE.MAR contains:

```
            .entry  setvalue,M<>

            movl    1,first
            movl    2,second
            movl    3,third
            ret

            .psect  example pic,usr,ovr,rel,gbl,shr,
                    noexe,rd,wrt,novec,long
first:      .blkl
second:     .blkl
third:      .blkl

            .end
```

Example 10-5: Sharing Data with a MACRO Program in a VAX-11 C Structure

The MACRO program initializes the locations first, second, and third in the psect named example and passes these values to the C program. The locations are referenced in the C program as members of the external structure named example.

Chapter 11

Global Symbols

In large programs, it is often desirable to share data among program modules by some means other than argument passing. In all C compilers, a variable to be shared by external functions must be declared with the **extern** keyword in each separately compiled function that refers to it. Each **extern** variable in VAX-11 C resides in its own program section. As illustrated in Chapter 10, **extern** variables are similar in this respect to FORTRAN named common blocks and to PL/I external variables.

Global symbols provide an alternative method for defining external variables and values. The keywords **globaldef** and **globalvalue** define objects that differ from **externs** both in their storage allocation and in their correspondence to elements of other languages. (These differences are spelled out explicitly in Table 11-1.) Global symbols provide a convenient and efficient way for a C function to communicate with assembly language programs, with VAX/VMS system services and data structures, and with other high-level languages that support global symbol definition, such as VAX-11 PL/I.

This chapter describes:

- The use of global symbols within C functions.
- The **globaldef**, **globalref**, and **globalvalue** keywords.
- The declaration and use of system-defined global symbols.

11.1 Global Symbols and extern Variables

Within VAX-11 C programs, you can define variables as global symbols when you are coding calls to system procedures. You can also use global symbols instead of **extern** variables to communicate between two or more VAX-11 C functions.

Table 11-1 summarizes the differences between global symbols and **extern** variables. Note that a primary difference is the manner in which the linker allocates storage. Linker storage allocation is described in more detail in Chapter 10.

Table 11–1: Comparison of Global Symbols and extern Variables

Global Symbol	extern Variable
Declared with the **globaldef**, **globalref**, or **globalvalue** keywords.	Declared with the **extern** keyword.
Corresponds to a global symbol declared in assembly language.	Corresponds to a FORTRAN common block.
Can be declared with **globalvalue** and does not occupy storage in program sections if expressible in 32 (or fewer) bits.	Always occupy storage in program sections.
No practical limit on the number of global symbols that can be defined and referenced in an object module.	Limited to approximately 65,532 **extern** names.

11.2 The globaldef and globalref Keywords

The **globalref** and **globaldef** keywords, respectively, declare and define a global variable. Global variables are exactly like **static** variables, except that their link-time scope is the entire program instead of a single compilation unit. If you do not specify a program section name, by **globaldef** {"name"}, VAX–11 C places **globaldef**'s definition for the name in a default program section. The definition is placed in the $CODE psect if it is defined with the **readonly** keyword; it is placed in the $DATA psect if it is *not* defined with the **readonly** keyword. Thus, **globaldef** avoids the use of named program sections by **extern** declarations, making the limited number of named program sections available for operations that require them.

The keywords **globaldef** and **globalref** are used similarly with external data definitions and **extern** declarations. That is, **globalref** is used to refer to storage allocated elsewhere (usually by a **globaldef** definition). For example:

In one compilation unit:

```
/* DEFINITION OF EXTERNAL VARIABLE: counter
   RESIDES IN A PROGRAM SECTION NAMED counter */
int counter = 0;

/* DEFINITION OF GLOBAL VARIABLE: velocity
   RESIDES IN THE PROGRAM SECTION $DATA */
globaldef double velocity = 3.0e10;

/* A C MAIN FUNCTION */
main()
{
.
.
}
```

In a separate compilation unit:

```
/* DECLARATION OF EXTERNAL VARIABLE:
   THE LINKER RESOLVES THIS REFERENCE
   TO THE PROGRAM SECTION counter */
extern counter;

/* DECLARATION OF GLOBAL VARIABLE:
   THE LINKER RESOLVES THIS REFERENCE
   TO THE PROGRAM SECTION $DATA */
globalref double velocity;

/* ANOTHER C FUNCTION THAT USES
   counter AND velocity */
fn(f)
{
.
.
}
```

Notice that initializers can appear in definitions of global variables (as in definitions of **extern** variables), but not in references to global variables. Initialization is possible only when storage is allocated for the object. This distinction is especially important when the **readonly** keyword is used; unless the global (or **extern**) variable is initialized when the variable is defined, its value is undefined.

NOTE

In the VAX-11 MACRO programming language, it is possible to give a global variable more than one name. However, in VAX-11 C, only one global name can be used for a particular variable. VAX-11 C assumes that distinct global names denote distinct objects; the storage associated with different names must not overlap.

11.3 The globalvalue Keyword

A variable declared with **globalvalue** does not require an address reference in storage. Instead, the compiler can refer to it by its value during execution. If an initializer appears with **globalvalue**, the name becomes a global symbol for the given initial value. If no initializer appears, the **globalvalue** construct is considered a reference to some previously defined global value. (Note that **globalvalue** can be used only with the data types **int** and **long**.)

Predefined global values serve many purposes in VAX/VMS system programming, such as the definition of status values (see Section 9.6.4).

Global values are useful because they allow many programmers in the same environment to refer to values by name, without regard to the actual value (for example, the integer) associated with the name. The actual values can change, as dictated by general system requirements, without affecting all the programs that use system resources. As mentioned in Chapter 9, it is customary in VAX/VMS system programming to avoid explicit references to such values as those returned by system services, and to use instead the global names for those values.

11.4 Enumerated Global Values

When the **globaldef** storage class keyword is used with an **enum** definition, the enumerated constants in the definition become **globalvalue**s, initialized as required to form a properly ordered list of the values. Variables of the enumerated type become **globaldef**s.

When **globalref** is used with **enum**, all enumerated variables are **globalref**s, and the enumerated constants refer to **globalvalue**s of the same names. For example:

In the first compilation unit:

```
/* DEFINE GLOBAL ENUMERATED TYPE */
globaldef enum light { dim,medium,bright } light_val;

main()

{
     light_val = dim;
     /* CALL FUNCTION */
     fnlv();
}
```

In the second compilation unit:

```
globalref enum light { dim,medium,bright } light_val;

fnlv()
{
     if (light_val < bright ) printf("TOO DIM\n");
}
```

In the first compilation unit, the **enum** definition establishes light_val as a **globaldef** of the enumerated type light. It also establishes the ordered list of enumerated **globalvalue**s dim, medium, and bright.

The **globalref** declaration in the second compilation unit allows the enumerated constants to be used as **globalvalue**s. That is, the constants can be referenced, but not initialized.

Chapter 12

Program Development

Throughout the process of VAX-11 C program development, you have to interact with the VAX/VMS operating system. Through this interaction, you either create or use many different types of files. Figure 12–1 shows the kinds of files required or created during program development, as well as the commands that relate to those files.

This chapter summarizes the following information about VAX/VMS:

- The rules for specifying input and output files for commands and programs.
- The commands available to you for file creation, modification, and maintenance.
- The use of command procedures as an aid to program development.
- The commands for creating and using text and object libraries.

For a tutorial introduction to these concepts, see the *VAX/VMS Primer*. For detailed definitions of commands and file specifications, see the *VAX/VMS Command Language User's Guide*.

12.1 File Specification Formats and Defaults

A file specification provides the system with all the information it needs to locate a unique file. In Figure 12–1, all input and output files are specified in their simplest form.

To define a unique C source file, you need only give it a unique name and a file type of C. All other portions of a file specification are allowed to default to system- and command-supplied names. For example, in Figure 12–1 the following defaults are in effect:

- All the commands shown use the current default device and directory to locate a specified file.
- The EDIT command does not assume any defaults. The file type C is specified in this example so that the file type can be defaulted for the CC command.

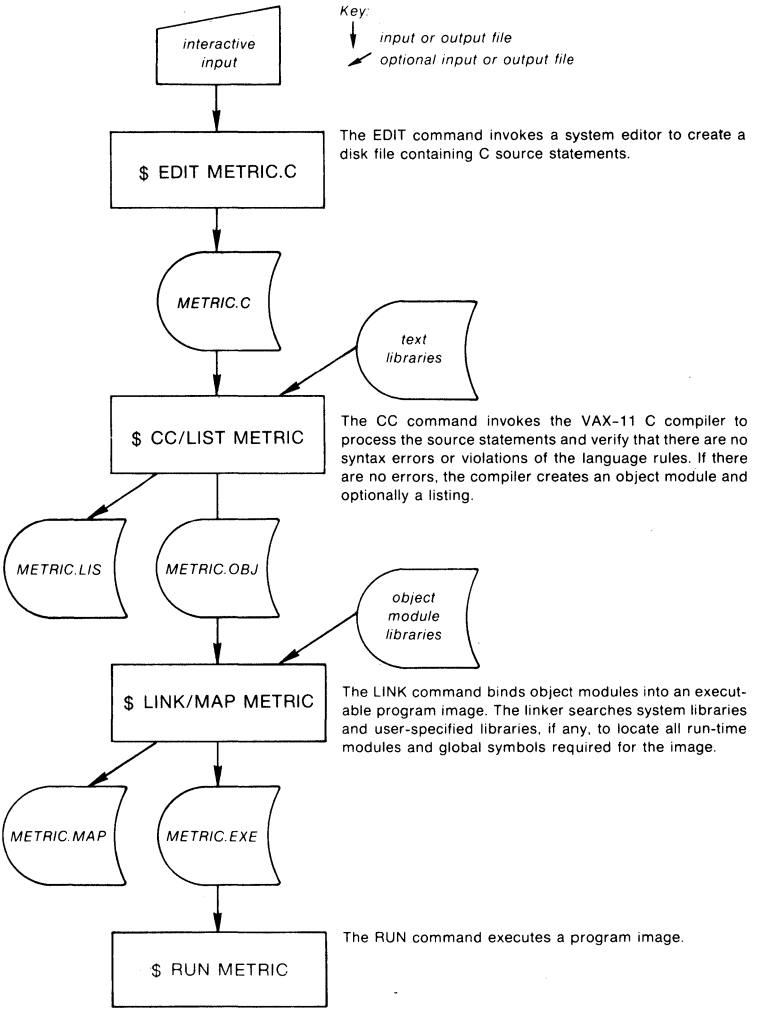


Figure 12-1: Commands for VAX-11 C Program Development

- The CC command assumes, if no file type is specified for a source file, that its file type is C. If no qualifiers override the default output file types used by CC, the compiler uses the default file types LIS and OBJ for the listing and object files, respectively.
- The LINK command assumes, if no file type is specified for an input file, that its file type is OBJ. If no qualifiers override the linker's default output file types, the linker uses the default file types EXE and MAP for the image and map files, respectively.
- The RUN command assumes, if no file type is specified, that the input file type is EXE.

Table 12–1 summarizes the syntax of VAX/VMS file specifications.

The following example shows a COPY command with a complete file specification:

```
$ COPY TUCSON::DBA3:[WILL]MEMO.DAT;3 HERE.DAT
```

This command copies the third version of the file MEMO.DAT in the directory [WILL] on the device DBA3 from the remote node TUCSON to the file HERE.DAT on the local system. The input file located at the node named TUCSON is fully specified. The output file HERE.DAT will be placed in the current default device and directory. If that directory does not contain a file named HERE.DAT, the COPY command will give the copied file a version number of 1. Otherwise, HERE.DAT will have a version number one greater than the highest version number of the existing file.

12.1.1 Temporary Defaults

Many VAX/VMS file-handling commands use temporary defaults under certain conditions. When a command such as PRINT or TYPE accepts a list of input file specifications, it uses explicit elements of one file specification as a temporary default for subsequent ones. Some examples follow.

```
$ PRINT [PROJECT.DATA]ALPHA,BETA.DAT,GAMMA
```

In this example, the PRINT command uses the default input file type LIS for the first input file and the file type DAT as specified for the second input file. It then applies the temporary default DAT to the file GAMMA. The PRINT command prints the highest existing versions of ALPHA.LIS, BETA.DAT, and GAMMA.DAT from the directory [PROJECT.DATA] on the current default device.

```
$ PRINT [PROJECT.DATA]FOREST.TXT,,DAT,,REF
```

Here, the PRINT command uses the temporary default FOREST as a file name and prints the files FOREST.TXT, FOREST.DAT, and FOREST.REF.

Temporary defaults are applied to device names, directories, file names, and file types. After the command is executed, the temporary defaults are no longer in effect.

Table 12-1: Summary of File Specification Syntax

Field	Syntax Rules	Defaults
node	1 - 6 characters terminated by ::	Local node
device dev	Valid mnemonic or logical name	SYS$DISK
c	A - Z	A
u	0 - 65535	0
directory [name] [name.name...]	1 - 9 characters up to 8 names, separated by periods (.)	Current default[1]
filename	0 - 9 characters	*Input:* temporary defaults apply[2] *Output:* same as input file
filetype	0 - 3 characters preceded by a period (.)	Applied by command; temporary defaults apply[2]
version	0 - 32767 preceded by a semicolon (;) or a period (.)	*Input:* highest[3] *Output:* highest + 1

1. [*] all directories
   [name...] all directories in path
   [*...] all subdirectories in all directories
   [-.name] back up a directory
2. * — all file names
   string — match all names containing "string"
   str%ng — match any character in % position
3. * — all versions
   ; — use most recent version

12.1.2 Changing the Default Directory

To change the default device or directory that is applied to all file specifications, use the SET DEFAULT command. Unless you override them in the explicit specification of a file, defaults set by this command remain in effect for all subsequent commands until you either issue a new SET DEFAULT command or log off the system.

For example:

```
$ SET DEFAULT [PROJECT.SOURCE]
$ CC METRIC
```

The CC command compiles the source program METRIC.C from the current default directory [PROJECT.SOURCE]. The output file, METRIC.OBJ, is also placed in this directory.

12.2 Logical Names

Another way to refer to a specific device, directory, or file is with a logical name. It can represent an entire file specification or the leftmost portion of one. To create logical names, use the DEFINE command. For example:

```
$ DEFINE SRC [PROJECT.SOURCE]
$ TYPE SRC:ALPHA.C
```

This command creates the logical name SRC to represent the directory specification [PROJECT.SOURCE]. When SRC is used with the TYPE command, the logical name in the file specification is replaced by its current equivalence name. The TYPE command displays the file [PROJECT.SOURCE]ALPHA.C.

Only one logical name is permitted in a file specification. It must be the first or only element, and it must be followed by a colon if any other elements are present.

The VAX/VMS system maintains tables of all logical names created by users. There are three kinds of logical name tables:

- **Process**. A separate logical name table exists for every user, or process, on the system. These names are available only to the user who defines them. A DEFINE command places a logical name in the process logical name table by default.
- **Group**. A separate logical name table exists for every group on the system. The names in any of these tables can be accessed only by users who have the same group number in their user identification code. To place a name in the group logical name table, you must specify /GROUP on a DEFINE command, and you must have the GRPNAM user privilege.

- **System**. There is a single system logical name table. The logical names in this table can be accessed by all users. To place a name in the system logical name table, you must specify /SYSTEM on a DEFINE command, and you must have the SYSNAM user privilege.

12.2.1 Logical Name Translation

When the system attempts to locate an equivalence name for the name of a C source file, or for a portion of a file specification, it is said to be performing a logical name translation. The system searches the process, then the group, then the system logical name tables. Each time the system translates a logical name, it examines the result to see if there is still a logical name. If so, it translates the result. This recursive translation occurs until the file specification is complete or until 10 recursive translations have been made.

You can determine the current equivalence for a logical name by entering the SHOW TRANSLATION command. For example:

```
$ SHOW TRANSLATION SRC
  SRC = "[PROJECT.SRC]"     (process)
```

The response gives the translation and indicates that the logical name SRC was found in the process logical name table.

A logical name assignment is deleted when a new definition is given for the name or when the name is explicitly deleted with a DEASSIGN command. For example:

```
$ DEASSIGN SRC
```

This command deletes the table entry for the logical name SRC.

12.2.2 Uses of Logical Names

VAX/VMS system programs use logical names in many ways. For example, the VAX-11 C compiler and the VAX/VMS Linker use logical names to provide default libraries for **#include** text modules and object module libraries, respectively.

Of principal interest to VAX/VMS programmers is the ability to use logical names to provide device and file independence when executing program images or command procedures. For example, the file specification associated with a file pointer in a C source program (as in the **fopen** function) can be a logical name. Each time you execute the program, you can issue a DEFINE command to provide a different equivalence name for the C file.

12.2.3 Commands to Control Logical Names

Table 12-2 lists the VAX/VMS command language, DCL, commands that maintain logical names.

Table 12–2: Commands for Maintaining Logical Names

Command	Function
DEFINE	Creates a logical name and places it in the process, group, or system logical name table. The /PROCESS, /GROUP, and /SYSTEM qualifiers specify the table in which the name is to be placed.[1]
DEFINE/USER	Creates a logical name for the execution of the next image only. The name is automatically deleted following the completion of the next command or program.[1]
ASSIGN	Provides the same function as DEFINE. However, the order of the command parameters is reversed.[1]
DEASSIGN	Deletes a logical name from the process, group, or system logical name table.[1]
SHOW TRANSLATION	Displays the result of translating a logical name once and displays the name of the table in which the logical name was found.[1]
SHOW LOGICAL	Displays the result of translating a logical name recursively. This command is performed by a separate program and causes the current image that is executing, if any, to be terminated.

1. This command is executed by the command interpreter and can be issued when a program is interrupted with CTRL/Y.

12.3 Creating and Maintaining Files

Table 12–3 describes some of the basic file-handling commands available to programmers in the VAX/VMS command language, DCL. For detailed descriptions, see the *VAX/VMS Command Language User's Guide*. For online assistance in entering a command or determining its parameters, qualifiers, or options, use the HELP command.

Table 12-3: VAX/VMS Commands for File Maintenance

Category	Command	Command Function
File creation	CREATE	Creates a file from records or data that follows in the input stream; for example, lines entered from a terminal or placed in a batch input file.
	EDIT	Invokes one of the VAX/VMS interactive editing programs, for example, SOS, EDT, or EDI.
Correcting and modifying files	EDIT	Invokes one of the interactive editors to make changes or additions to a disk file.
	SET DIRECTORY	Modify the characteristics of a directory.
	SET FILE	Modify the characteristics of a file.
Cataloging and organizing files	CREATE/DIRECTORY	Establishes a new directory or a hierarchy of directories to catalog files.
	DIRECTORY	Lists files and information about them. Can list files with common file names or file types, files in one or more directories, files created since a certain date, and so on.
	LIBRARY	Creates and maintains libraries of **#include** text modules and libraries of object modules.
	RENAME	Changes the directory in which a file is cataloged; or changes the file name, file type, or version number of a file.
	SET DEFAULT	Changes the current default device or directory.
Copying and backing up files	{ ALLOCATE INITIALIZE MOUNT }	Provide device-handling and control commands that let you access data written on nonsystem disks, on magnetic tapes, or on punched cards; or to output data to a disk or tape.

Table 12-3: (Cont.) VAX/VMS Commands for File Maintenance

Category	Command	Command Function
	COPY	Copies the contents of a file or files to another file or files.
Deleting files	DELETE	Makes the contents of a file inaccessible by removing its directory entry.
	PURGE	Deletes a specified number of earlier versions of a file or a group of files.

12.4 The HELP Command

Many of your questions about VAX/VMS commands and VAX-11 C can be answered by the HELP utility. The HELP utility is a tree-structured group of specially formatted text files and a program that allows you to display these files at the terminal. Each level of the tree displays the information for that level plus a list of topics that are available at the next lower level of the tree.

The format of the HELP command is as follows:

```
HELP [subtopic [subtopic [...] ] ] RET
```

The VAX-11 C help file contains information about the CC command line, the C language, and the VAX-11 C compiler diagnostic messages.

You can obtain the list of the VAX-11 C help topics at the top level of the tree by typing:

```
$ HELP CC
```

12.5 Using Command Procedures

A command procedure is a file that contains a sequence of VAX/VMS commands and, optionally, data. You can cause the commands in the procedure to be executed in either of two ways:

- Interactively, you specify the name of the file following the @ (execute procedure) command. For example:

```
$ @TESTAM
```

The @ command assumes that the file type of a command procedure is COM. Thus, the @TESTAM command executes the procedure TESTAM.COM.

- With DCL's SUBMIT command, you can submit the command procedure to a system batch job queue for execution. For example:

```
$ SUBMIT TESTAM
```

This command enters the file TESTAM.COM in the system batch job queue. On completion of the job, the system prints a log file that indicates how the job ran.

You can devise and use command procedures to simplify and enhance your program development. For example, you can write a command procedure that will compile, link, and run a specific C program. The procedure can specify all the libraries needed by the CC and LINK commands and even contain all the input data you would require to test the program.

Command procedures can also be generalized. By taking advantage of such DCL features as the assignment statement and the IF, GOTO, and ON commands, you can write a command procedure that looks like a program; it can process variables, make decisions based on their values, and perform error condition handling.

Example 12–1 suggests a way to construct command procedures for C program development and testing.

```
$ ON WARNING THEN EXIT                                          ❶
$ !                                                             ❷
$ LIST := ""                                                    ❸
$ !
$ IF P1 .EQS. "L" THEN LIST := /LIST=LP:                        ❹
$ CC'LIST' APPLIC,METRIC+DATAB/LIBRARY                          ❺
$ LINK APPLIC,METRIC,[APPLICLIB]APPLIC/LIBRARY
$ !
$ RUN APPLIC                                                    ❻
55555.88888888
4444.333
22
'alphabetic string'
$ EXIT
```

Example 12–1: A Sample Command Procedure

The following notes are keyed to the circled numbers in Example 12–1.

❶ The ON command establishes an error-handling routine for the command procedure. This command specifies that the procedure is to exit if any error occurs with a severity of warning or greater. Thus, if the CC command returns with an unsuccessful status, the procedure executes an EXIT command that causes the procedure to terminate immediately. Otherwise, the procedure continues with the LINK command. If that command issues a warning or error, the procedure exits. Otherwise, the RUN command executes.

❷ The exclamation point (!) is a comment delimiter.

❸ The procedure creates a variable, or command symbol, named LIST and gives it an initial value of a null string. The assignment statement := gives a symbol a character-string value.

❹ The procedure then tests whether any values were specified when the procedure was invoked. A value passed to a command procedure, that is, a parameter, is given a default name of P*n*, where *n* is the position of the parameter in the command. For example, the procedure may be executed as follows:

```
$ @TESTAM L
```

In this execution, L is the first (and only) parameter. The symbol P1 is given a value of L. In this procedure, L indicates that a listing file is requested. The IF command tests the value of P1. If P1 is L, then the symbol LIST is redefined to have a value of /LIST=LP:.

❺ Following the CC command name, the symbol LIST is specified inside apostrophes. The apostrophes, in this context, are substitution operators that request the command interpreter to substitute the symbol LIST with its current value. If LIST is a null string (that is, if L was not specified), the command after substitution is:

```
$ CC APPLIC,METRIC+DATAB/LIB
```

If L was specified, the command after substitution is:

```
$ CC/LIST=LP:  APPLIC,METRIC+DATAB/LIB
```

❻ The program APPLIC is executed. It reads input data from the default input device. When a command procedure executes, the default input device is the command procedure itself. Thus, the data are read from the procedure file. In a command procedure, any line that does not begin with a dollar sign ($) is treated as input data for the previous command or program. The input terminates (and an actual end-of-file condition occurs) when a new line that begins with a dollar sign ($) is encountered. In this example, the program APPLIC reads all the lines between the RUN command and the EXIT command.

For more detailed information on the commands shown in the preceding example, and for additional examples of techniques you can use in command procedures, see the *VAX/VMS Guide to Using Command Procedures*.

12.6 Libraries

Libraries collect frequently used text or functions in easily accessible modules. Libraries make program development efficient in a number of ways.

With libraries, you can avoid programming at a machine-language or machine-dependent level. Within reason, the text and object libraries supplied with VAX-11 C allow you to use your existing programming methods (and, in many cases, existing source text) to develop C programs for the VAX-11 computer, without detailed knowledge of the VAX-11 architecture or VAX/VMS operating system.

In addition, most portability problems occur below the level of the C language and can be addressed by changing (or creating) a library, thereby modifying an entire set of programs that use the same features. Programs then move more easily from one computer system to another.

Libraries help you avoid the pitfall of "reinventing the wheel." Once a function or other item has been shown to be correctly designed and implemented, you can put it in a user library, perhaps for use by several programmers in the same environment. For example:

- Commonly used pieces of source text can be grouped into modules in a text library. The source text need not comprise entire C functions. In fact, text library modules are used frequently in C programs to supply complicated definitions, macros, or text substitutions needed by the program. You select a module from a text library by writing an **#include** control line in the program, specifying the module's name. (See Chapter 7 for an explanation of the forms of the **#include** control line.) You then specify the name of the library with a qualifier (/LIBRARY) to the CC command.
- Commonly used object (compiled) code can be grouped into modules in an object library. Again, the object modules need not comprise entire C programs, although each results from a module of source text that can be compiled separately. Object modules are located and selected automatically by the VAX/VMS Linker to resolve references in the program to otherwise undefined functions or global symbols. If the object modules are located in a library that you have created, you must specify the library's name with a qualifier to the LINK command.

Libraries can simplify the user's interface to the program development commands, CC and LINK. Both commands search a series of default user libraries and DIGITAL-supplied libraries for unresolved references. This feature allows a new VAX/VMS programmer to use the simplest forms of the CC and LINK commands, with reasonable assurance that a correctly written C program will compile and execute on a VAX/VMS system. That is:

- The library of source text (SYS$LIBRARY:CSYSDEF.TLB) supplied with the VAX-11 C compiler is searched for references that are still unresolved after a search for any specified or default text libraries has been made.
- You can define equivalents for the logical names C$LIBRARY and LNK$LIBRARY. These logical names describe the defaults for user-defined text and object libraries, respectively. If the CC command cannot resolve references from **#include** lines in the text libraries you have specified, it searches the equivalent of C$LIBRARY for the necessary text modules. If the LINK command cannot resolve the function references in the object libraries you specify, it searches the equivalent of LNK$LIBRARY. The library SYS$LIBRARY:CRTLIB.OLB, supplied with the VAX-11 C compiler, contains the object code for VAX-11 C library functions (the functions described in Chapter 6). You must specify SYS$LIBRARY:CRTLIB.OLB explicitly in the LINK command, or assign it to the logical name LNK$LIBRARY, in order to resolve references to these functions.
- The VAX/VMS system libraries (VMSRTL.EXE and STARLET.OLB) call routines in the VAX-11 Common Run-Time Procedure Library. (These routines are documented in the *VAX-11 Run-Time Library Reference Manual.*) Any references to functions that are still unresolved after a search for the libraries specified in the LINK command and the equivalent for LNK$LIBRARY are assumed to be calls to these run-time procedures. These calls can result from an explicit function reference in the program, or they can be internal calls generated by the compiler to perform common operations such as input and output, calls to mathematical functions, and so forth. The LINK command automatically searches VMSRTL.EXE and STARLET.OLB to resolve these references.

The remainder of this section describes how to create text libraries and object libraries. This description is confined to areas that are of particular interest to C programmers. For more detailed coverage of the topics presented here, consult the following VAX/VMS manuals:

- *VAX-11 Utilities Reference Manual* — for information on the VAX/VMS Librarian.
- *VAX-11 Guide to Creating Modular Library Procedures* — for more extensive information on modular programming techniques for VAX/VMS.
- *VAX-11 Run-Time Library Reference Manual* — for instructions on making direct calls to procedures in the VAX-11 Common Run-Time Procedure Library.

12.6.1 Text Libraries

A text library is a file that contains text organized into modules and a table indexing the modules. The LIBRARY command creates and modifies text libraries; these libraries have a default file type of TLB. To use libraries of C **#include** modules, you must:

1. Create one or more libraries consisting of C source text.
2. Specify the name of a module in an **#include** control line in the C source program.
3. Specify the name of the library in the CC command to compile the source program, or define the library as a default user library with an equivalence name for C$LIBRARY.

Figure 12–2 illustrates the creation of an **#include** module library and its use in compiling C programs.

12.6.1.1 Naming Text Modules

When the LIBRARY command adds a module to a library, it uses by default the file name of the input file as the name of the module. In the example in Figure 12–2, the LIBRARY command adds the contents of the files APPLIC.SYM and DECLARE.C to the library and names the modules APPLIC and DECLARE.

Alternatively, you can use the /MODULE qualifier to specify a name for a library module. For example:

```
$ LIBRARY/TEXT/INSERT CCFILES -
$_DECLARE.C/MODULE=EXTERNAL_DECLARATIONS
```

This command inserts the contents of the file DECLARE.C into the library CCFILES and names the module EXTERNAL_DECLARATIONS. This module can be included in a C source file with the control line:

```
#include external_declarations
```

Table 12–4 summarizes the commands that create libraries and provide maintenance functions. For a complete list of the LIBRARY command qualifiers and for a description of other DCL commands listed in Table 12–4 see the *VAX/VMS Command Language User's Guide*.

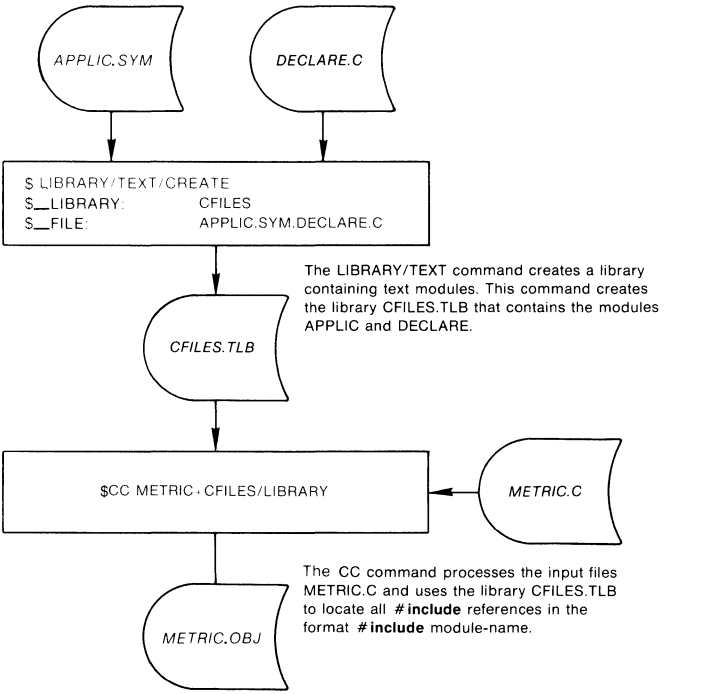


Figure 12–2: Creating and Using an #include Module Library

Table 12-4: Commands to Control Library Files

Function and Command Syntax[1]
Create a library. $ LIBRARY/TEXT/CREATE *library-name file-spec,...*
Add one or more modules to a library. $ LIBRARY/TEXT/INSERT *library-name file-spec,...*
Replace one or more modules in a library. $ LIBRARY/TEXT/REPLACE[2] *library-name file-spec,...*
Specify the names of modules to be added to a library. $ LIBRARY/TEXT/INSERT *library-name file-spec*/MODULE=*module-name*
Delete one or more modules from a library. $ LIBRARY/TEXT/DELETE=*(module-name,...) library-name*
Copy a module from a library into another file. $ LIBRARY/TEXT/EXTRACT=*module-name library-name*
List the modules in a library. $ LIBRARY/TEXT/LIST/OUTPUT=*file-spec library-name*
Rename a library or move a library to another directory. $ RENAME *old-library-name new-library-name*
Delete a library. $ DELETE *library-name*
Copy or backup a library. $ COPY *input-library-name output-library-name*

1. The LIBRARY command qualifier /TEXT indicates a text module library. By default, the LIBRARY command assumes an object module library.

2. REPLACE is the default function of the LIBRARY command, if no other action qualifiers are specified. If no module exists with the given name, /REPLACE is effectively /INSERT.

12.6.1.2 Default C Libraries

You can define one of your private **#include** module libraries as a default library. The C compiler searches the default library after it searches libraries specified in the CC command.

To define a default library, make the library file specification equivalent to the logical name C$LIBRARY, as in the following example:

```
$ DEFINE C$LIBRARY SYS$LOGIN:DATAB.TLB
```

While this assignment is in effect, the compiler automatically searches the library SYS$LOGIN:DATAB.TLB for any **#include** modules that it cannot locate in libraries explicitly specified in the CC command.

You can define the logical name C$LIBRARY in the process, group, or system logical name table. If the name is defined in more than one table, the C compiler uses the equivalent for the first match it finds in the normal order of search (that is, first the process, then the group, then the system table). Thus, if C$LIBRARY is defined in both the process and group logical name tables, the process logical name table assignment overrides the group logical name table assignment.

12.6.1.3 Default System #include Library

When it cannot find **#include** modules in libraries specified in the CC command or in the default library defined by C$LIBRARY, the C compiler searches the library identified by the name:

```
SYS$LIBRARY:CSYSDEF.TLB
```

CSYSDEF.TLB is a library of **#include** modules supplied with VAX-11 C. It contains declarations of values returned by the VAX/VMS system services, as well as the text of all files of type h that are supplied with VAX-11 C. For example, you can include the standard I/O definitions in a program with either of these **#include** lines:

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

which includes the file SYS$LIBRARY:STDIO.H; or equivalently

```
#include stdio
```

which includes the text module STDIO from SYS$LIBRARY:CSYSDEF.TLB.

12.6.2 Object Libraries

An object library is a file of object code organized into modules. The modules are indexed by two tables:

- A module name table, which lists the names of the modules in the library. The names are those given to the modules when they are compiled.
- A global symbol table, which lists all the global symbols defined in each module.

These are the tables that the linker searches.

12.6.2.1 Creating An Object Module Library

The LIBRARY command creates and updates libraries. It assumes by default that a library upon which it is performing a function is an object module library. You can use object module libraries to:

- Catalog and group commonly used functions.
- Provide a default set of modules for the linker to use in resolving global references in object modules it is linking.
- Enhance the performance of linking operations by putting all needed modules in a single library, thus reducing the number of files that need to be opened during linking.

Figure 12–3 illustrates the steps you would take to create object modules, to create a library, and to use the library when linking programs.

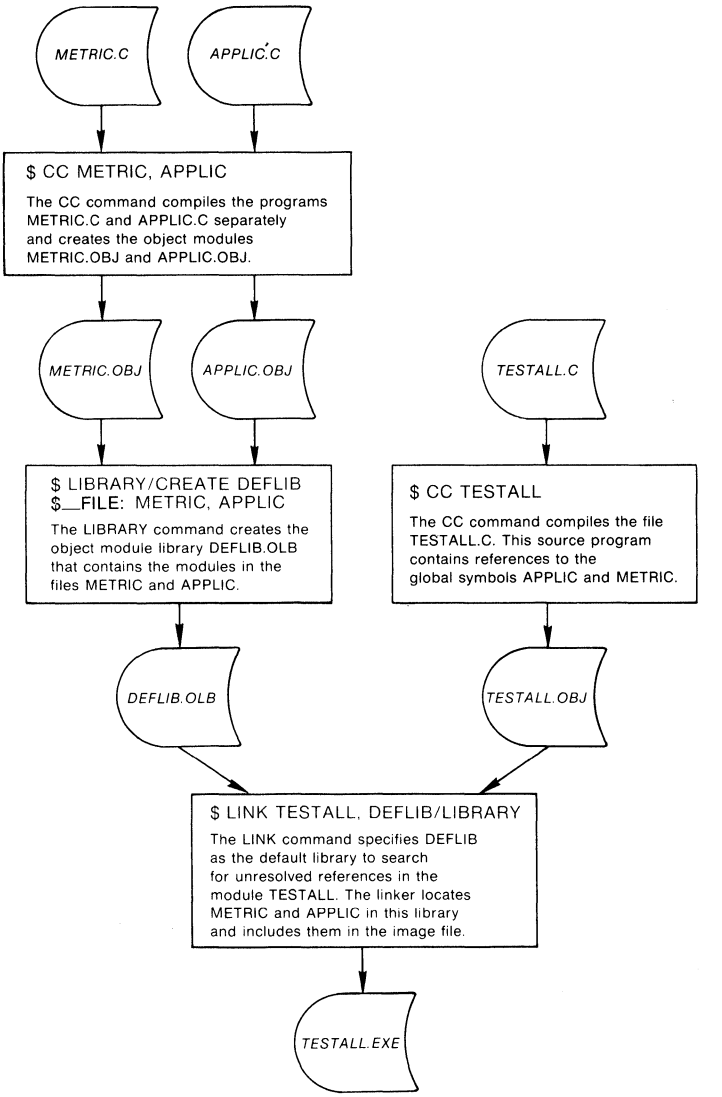


Figure 12–3: Creating and Using an Object Module Library

The LIBRARY command uses the following default file types:

- OLB indicates an object module library file.
- OBJ indicates an object module file.

When the LIBRARY command inserts an object module in a library, it:

- Enters the name of the module in the library's module name table.
- Enters all global symbols from the object module in the library's global symbol table.

For example, a C program named QUEUES.C might contain the following definitions:

```
ready()
{
globalref char *zp;
.
.
}

globaldef char *stk;

addel(queue,point)
char *queue,*point;
{
.
.
}

remvel(queue,point)
char *queue,*point;
{
.
.
}
```

This module can be compiled and placed in a library as follows:

```
$ CC QUEUES
$ LIBRARY/INSERT DEFLIB QUEUES
```

After the LIBRARY command in this example has been executed, the module name table for the library DEFLIB.OLB contains an entry for the module named QUEUES. The library's global symbol table contains entries for the names ready, addel, remvel, and stk. Object modules that refer to any of those names can then be linked with this library. When the library is specified as input to the linker, the linker searches the library's module name table and global symbol table for unresolved references.

12.6.2.2 Default User Object Module Libraries

You can define one or more of your own object module libraries as default user libraries. The linker searches them for unresolved references after it searches modules and libraries specified in the LINK command.

To indicate that a library is a default user library, enter a DEFINE command as shown in the following example:

```
$ DEFINE LNK$LIBRARY DBA5:[MY.LIBS]DEFLIB
```

LNK$LIBRARY is a logical name; DBA5:[MY.LIBS]DEFLIB is the name of an object module library that you want the linker to search automatically in all subsequent link operations.

You can establish any object module library as a default user library by creating a logical name for the library. The logical names you must use are LNK$LIBRARY (as above), LNK$LIBRARY_1, LNK$LIBRARY_2, and so on, to LNK$LIBRARY_999. If more than one of these logical names exists when a LINK command executes, the linker searches them in numerical order, beginning with LNK$LIBRARY.

Default libraries may also be shared by users who have the same group number in their user identification codes. To share a library in a group, make the logical name assignment in the group logical name table (the GRPNAM user privilege is required for this assignment). For example:

```
$ DEFINE/GROUP LNK$LIBRARY [APPLICLIB]APPLIC
```

When this logical name assignment is in effect, the linker searches the library [APPLICLIB]APPLIC.OLB, as necessary, for all link operations performed by processes in the group of the user who entered this command.

The logical names LNK$LIBRARY through LNK$LIBRARY_999 can exist in both process and group logical name tables without conflict. Similarly, these names can also be defined in the system logical name table to provide a default system-wide user library (the SYSNAM user privilege is required).

12.6.2.3 System Libraries

The directory identified by the system-defined logical name SYS$LIBRARY contains the library files:

- VMSRTL.EXE
- STARLET.OLB

The file VMSRTL.EXE contains the VAX-11 Run-Time Library. The procedures in this library provide:

- Commonly used mathematical and string-handling functions.
- Procedures that support code produced by VAX/VMS compilers.

VMSRTL.EXE is a library in shareable image format; that is, it is prelinked and can be accessed concurrently by many images. The procedures in a shareable image library can be used by a program even though the procedures are not physically included in the program image. The references to the procedures in the shareable image library are not resolved until the program actually runs. For information about creating shareable image libraries, and a description of the VAX-11 Run-Time Library, see the *VAX-11 Run-Time Library Reference Manual.*

STARLET.OLB contains, in object module form, all the procedures in VMSRTL.EXE, as well as additional run-time modules required by various compilers and system programs. The global symbols for VAX-11 C have nonstandard names for compatibility with other C implementations. Therefore, the VAX-11 C object modules cannot be included in STARLET.OLB; they are in a separate library (SYS$LIBRARY:CRTLIB.OLB).

By default, the linker searches STARLET.OLB and VMSRTL.EXE to resolve references to external names that are still unresolved after the libraries specified in the LINK command and the user default libraries have been searched.

You can control whether the linker searches VMSRTL.EXE and STARLET.OLB by using the /NOSYSSHR and /NOSYSLIB qualifiers. These qualifiers have the following effects:

- If you specify /NOSYSSHR, the linker does not search the shareable version of the Run-Time Library. Any run-time procedures required by your program will be included in your image from STARLET.OLB.
- If you specify /NOSYSLIB, the linker searches neither VMSRTL.EXE nor STARLET.OLB. You must provide private versions or copies of any run-time procedures required by your program.

For example:

```
$ LINK/NOSYSSHR METRIC,APPLIC,DEFLIB/LIBRARY
```

In this link operation, the linker searches the library DEFLIB.OLB, then any default user libraries, then STARLET.OLB. It does not search VMSRTL.EXE.

For additional information on the LINK command, see Chapter 14. For information on how to call Run-Time Library procedures and VAX/VMS system services from a C program, see Chapter 10.

Chapter 13

Creating Source Programs

The first step in developing a VAX-11 C program consists of creating the program's source file. VAX/VMS offers two supported text editors that allow you to do this: SOS and EDT. This chapter provides an introduction to the use of EDT. For information on SOS, refer to the *VAX-11 Text Editing Reference Manual.*

There are three other sources of information on EDT. The first is the *VAX-11 EDT Editor Reference Manual.*[1] The second is the computer-assisted course titled "Introduction to the EDT Editor" supplied with the VAX/VMS operating system. The third is EDT's help facility.

13.1 Introduction to EDT

EDT, the DEC Standard Editor, is an interactive general-purpose text editor. It offers two modes of operation: line editing, in which operations are performed on entire lines of text; and character editing, in which operations are performed on characters and words as well as on lines. Line editing is possible at either hard-copy or video terminals. Character editing, while usable at hard-copy terminals, is most effective at video terminals.

Line editing mode, with its English-like commands, is simple for the inexperienced user to learn. Character editing mode, while requiring practice, is also very simple. Therefore, EDT is a good editor for someone who must learn a text editor quickly.

EDT also offers many advanced features:

- Multiple text buffers. By default, editing operations take place within a single text buffer called MAIN. However, you can maintain an unlimited number of alternate text buffers as "holding areas" for text that you do not necessarily wish to incorporate in the output file.

1. Some installations may have the *EDT Editor Manual* instead of the *VAX-11 EDT Editor Reference Manual.* The two manuals contain the same information about EDT.

- Flexible input and output commands. You can copy files into an EDT text buffer after beginning the editing session, and you can output text buffers or portions of text buffers to files before ending the session.
- Macro capability. You can create sequences of line editing commands that you invoke with a single command.
- The ability to define keys for custom character editing applications. For example, a keypad key can be defined so that it inserts a specified line of text each time it is pressed. This function is especially useful in programming applications where certain statements may be repeated frequently.

Finally, EDT protects your text. Should your editing session end in an unexpected manner, you can recover all your editing operations by reentering the EDT command line with the /RECOVER qualifier. EDT then "replays" your editing session up to the point of interruption, using the contents of the journal file that it maintained during the lost session.

The following subsections introduce EDT's line editing commands and help facilities.

13.1.1 Line Editing Command Summary

When you invoke EDT, and throughout your editing session, EDT prompts you to enter line editing commands by displaying an asterisk. For example:

```
$ EDIT/EDT METRIC.C
    1               main()
*
```

Table 13–1 describes briefly in alphabetic order the most useful commands that you can enter in response to the line editing prompt (*). Examples of these commands occur throughout Sections 13.2, 13.3, and 13.4. The smallest acceptable abbreviation for each command is shown in bold type in the table.

All line editing commands are terminated with a RET. Most of the commands allow or require you to specify a range or ranges; the range specification tells EDT where the action of the command should take place. Section 13.4.1 summarizes range specifications, and the command examples show various ways of specifying a range.

Table 13–1: Summary of Line Editing Commands

Command	Function
CHANGE [range]	Invokes character editing mode for specified buffer
COPY [range1] **TO** [range2] [/**Q**UERY]	Copies lines specified by range1 to a location in an EDT buffer specified by range2; does not delete lines from original location
DEFINE { KEY / **M**ACRO }	Defines a new or revised key function for character editing mode, or defines a macro name
DELETE [range] [/**Q**UERY]	Deletes a specified line or lines
EXIT [file-spec]	Terminates EDT, saving the contents of the text buffer MAIN as the output file
FIND range	Moves the current line to a specified line
HELP [topic ...]	Displays information on a specified EDT command or function
INCLUDE file-spec [range]	Copies an external file to a location in a text buffer specified by range
INSERT [range]	Opens a text buffer for the insertion of text at the location specified by range
MOVE [range1] **TO** [range2] [/**Q**UERY]	Moves lines specified by range1 to the location specified by range2, deleting the lines from the source location
QUIT [/**SA**VE]	Terminates EDT without creating an output file, optionally saving the journal file
REPLACE [range]	Deletes specified lines from a text buffer and leaves the buffer open for insertion of text
SET [parameter]	Sets a variety of editor operating parameters
SET [**NO**]**NUMBERS**	Enables/disables the display of line numbers

Table 13-1: (Cont.) Summary of Line Editing Commands

Command	Function
SHOW [parameter]	Displays specified editor operating parameters
SUBSTITUTE /string1/string2/[range] [/**Q**UERY]	Replaces string1 with string2, either in the current line or in the specified range
[**SUBSTITUTE**] **NEXT** [/string1/string2]	Replaces string1 with string2, based either on the strings specified or on the previous SUBSTITUTE command
[**TYPE**] range	Displays specified lines and makes the first line in range the current line; the default command
WRITE file-spec [range]	Moves a copy of specified text from a buffer to a file

13.1.2 The Help Facilities

EDT offers online help in both line and character editing modes. In line editing mode, you invoke the help facility by entering the HELP command. Issued without parameters, this command displays information on how to get further help, plus a list of subjects for which help is available. If you enter one of the subjects as a parameter to the HELP command, EDT displays information on that subject, and possibly another list. For example:

```
*HELP DELETE

DELETE

The DELETE (abbreviation: D) command deletes the line
specified
    .
    .
Additional information available:
```

```
/QUERY
*HELP DELETE /QUERY

DELETE

/QUERY
    .
    .
        Q    Quit, do not delete any of the rest of the
             lines
        A    All, delete all of the rest of the lines
*
```

In character editing mode, you obtain help by pressing the HELP key on the keypad; EDT displays a diagram of the keypad with all the key functions identified. You can then obtain help on an individual function by pressing the key that invokes that function. (Section 13.5 shows how to find the HELP key.)

13.2 Invoking and Terminating EDT

An editing session begins when you invoke EDT with the EDIT/EDT command, and ends when you terminate EDT with the EXIT or QUIT command. You may start an editing session with no file and create the text for the file during the course of the session. Or you may specify an existing file when you start the session, in which case EDT loads the file into its MAIN text buffer. EDT does not destroy the contents of any existing file that you edit; it simply produces a new version, leaving the old version intact.

13.2.1 Invoking EDT

To invoke EDT, issue an EDIT/EDT command in the format:

EDIT/EDT[/qualifier...] file-spec

Qualifiers	Default
/[NO]COMMAND[=file-spec]	/COMMAND=EDTINI.EDT
/[NO]JOURNAL[=file-spec]	/JOURNAL=infile-name.JOU
/[NO]OUTPUT[=file-spec]	/OUTPUT=infile-spec
/[NO]READ_ONLY	/NOREAD_ONLY
/[NO]RECOVER	/NORECOVER

file-spec

Specifies the file to be created or edited. If the file does not exist, EDT creates it.

EDT does not provide a default file type. If you do not specify one, the file type is null.

/OUTPUT[=**file-spec**]
/NOOUTPUT

Supplies an alternate file specification for the output file. By default, EDT creates an output file upon exit that has the same name and type as the input file and a version number of 1 (if the input file does not exist) or one higher than the highest existing version (if the input file does exist).

If you specify /NOOUTPUT, EDT does not automatically create an output file when you issue the EXIT command.

The remaining qualifiers, which describe specialized editor functions, are described elsewhere: the /COMMAND qualifier, in Section 13.7.3; the /JOURNAL, /READ_ONLY, and /RECOVER qualifiers, in Section 13.6.

For convenience, you can issue the following command to equate a short command symbol (EDT, in this example) to EDIT/EDT:

```
$ EDT :== "EDIT/EDT"
```

After you issue this command, the command interpreter will recognize the symbol EDT (or whatever symbol you specify) as equivalent to EDIT/EDT.

When you invoke EDT, the response varies depending on whether or not the file that you specify exists. (Other factors, such as commands contained in a start-up command file named EDTINI.EDT, may further alter the response.) If the file does not exist, EDT so informs you, and prompts you to issue editing commands:

```
$ EDIT/EDT METRIC.C
Input file does not exist
[EOB]
*
```

The asterisk (*) is EDT's line editing prompt. When EDT is displaying the asterisk prompt, you can enter any of the commands listed in Table 13–1.

If the file exists, its first line is displayed instead of [EOB]:

```
$ EDIT/EDT METRIC.C
    1           main()
*
```

NOTE

If you invoke EDT and it does not display an asterisk prompt, you cannot enter line editing commands. This condition can result when the current default directory contains a start-up command file named EDTINI.EDT that causes EDT to enter character editing mode directly. If this happens, you can enter line editing mode by typing a CTRL/Z. You can override the unwanted effects of a start-up command file by including the /NOCOMMAND qualifier on the command line.

13.2.2 Terminating EDT

Use the EXIT command to terminate EDT and create an output file from the contents of the MAIN text buffer. To override the default output file, you can specify an output file with the EXIT command, as shown in the following example:

```
*EXIT ALTNAME.C
_DB1:[PROJECT]ALTNAME.C;1 55 lines
$
```

The QUIT command terminates EDT without creating an output file. You can use QUIT if you are simply reading a file without modifying it or if you do not want to save your edits.

13.3 Creating a New File in Line Mode

To create a new file, you issue an EDIT/EDT command that specifies a file that does not currently exist in your directory. After EDT responds with the asterisk prompt, issue the INSERT command (abbreviation I) followed by RET. The cursor or print head then moves to the right 16 spaces; this space is left by EDT to accommodate line numbers, although none appear at this stage. You can now enter as many lines of text as you wish. When you are finished entering text, terminate the insert with CTRL/Z. The following example illustrates this process:

```
$ EDIT/EDT EXAMPLE.TXT
Input file does not exist
[EOB]
*I
                This is the first line of EXAMPLE.TXT
                This is the second line of EXAMPLE.TXT
                This is the third line of EXAMPLE.TXT
                This is the fourth line of EXAMPLE.TXT
                This is the fifth line of EXAMPLE.TXT
                This is the sixth line of EXAMPLE.TXT
                This is the seventh line of EXAMPLE.TXT
                CTRL/Z ^Z
*
```

The [EOB] designation indicates that you are currently at end-of-buffer, and that any text you insert will be the only text in the buffer.

If you do not want EDT to leave space in front of each line for line numbers, you can issue the SET NONUMBERS command; EDT then begins each line at the left margin of the terminal. EDT continues to number lines, but does not display the numbers. You can restore the line number display later by issuing a SET NUMBERS command.

13.4 Editing an Existing File in Line Mode

To edit an existing file in your directory, issue an EDIT/EDT command that specifies its name. (For information on how to edit a file from a directory other than your own, see Section 13.4.8.) EDT displays the first line in the file, as shown in the following example:

```
$ EDIT/EDT EXAMPLE.TXT
    1            This is the first line of EXAMPLE.TXT
*
```

The number 1 to the left of the line is the line number. It is not part of the file. The file starts with the word *This*.

The line displayed is the current line. EDT uses the current line as the default in many of its operations. For example, an INSERT command that does not specify a range causes EDT to insert text in front of the current line.

The concept of "range" is central to all EDT line editing operations. The next section describes ways of specifying range. The sections that follow it describe the most common and useful line editing operations.

13.4.1 Range Specifications

A range consists of the line or lines on which EDT performs an operation. A range specification is a description of a range in terms that EDT can understand. All the line editing commands (except SUBSTITUTE NEXT) described in the sections that follow accept one or more range specifications, although many do not require one.

The simplest range specification identifies a single line of text. A line can be located by its position in the file relative to the current line, by a text string, or by its line number. Since line numbers are primarily useful in range specifications, they are described here.

When you insert lines of text in a new file, or when EDT loads an existing file into its MAIN buffer, each line of the file receives a number. The numbering starts with 1 and proceeds upwards by ones. If you insert lines of text between existing lines, EDT numbers the new lines using appropriate decimal increments. This technique ensures that there will be enough unique line numbers to cover any reasonable editing operation. EDT displays the line numbers whenever it displays

text, unless you have issued the SET NONUMBERS command. In that case, EDT does not display line numbers, but it does continue to assign them.

Single-line range specifications are listed in Table 13–2; examples appear below.

Table 13–2: Single-Line Range Specifications

Specification	Meaning
.	The current line
number	The line specified by the number
'string' or "string"	The next line containing the string you specify
-'string' or -"string"	The preceding line containing the string you specify
[range] { + / - } [number]	The line that is the specified number of lines after (or before, if minus) the single line specified by range (range defaults to the current line; number defaults to 1)
BEGIN	The first line in the text buffer
END	An empty line (designated by [EOB]) following the last line of text in the text buffer.

Specification	Meaning
20.6	The line numbered 20.6
"#include"	The next line that contains the string *#include*
-"putchar(c);"	The first preceding line that contains the string *putchar(c);*
-6	The line six lines before the current line
'i++'+4	The line four lines after the line that contains the string *i++*

When EDT searches for a string, the case of the search string need not match the case of the target. For example, *getchar* is a match for *GETCHAR* or *Getchar*. This condition is the default; you can change it with the SET SEARCH command.

There are several methods available for specifying a range of more than one line. They are listed in Table 13–3; examples appear below.

Table 13-3: Multiple-Line Range Specifications

Specification	Meaning
[range1] { : / THRU } [range2]	The set of lines from range1 through range2, which are single-line range specifications (both range1 and range2 default to the current line, if omitted)
[range] { # / FOR } number	The specified number of lines beginning with the single line specified by range (range defaults to the current line, if omitted)
BEFORE	All lines in the buffer that precede the current line
REST	The current line and all lines in the buffer that follow it
WHOLE	The entire buffer
range, range... or range AND range AND...	All lines specified by each single-line range
[range] ALL 'string'	All lines in the range containing the specified string (the default for range is the entire buffer)

Specification	Meaning
2:6.5	Lines 2 through 6.5, inclusive
'#include'#5	The line containing the string *#include* and the four lines following it, for a total of five lines
.-10:.	The line 10 lines before the current line through the current line, inclusive
10:50 ALL 'get'	All lines from line 10 through line 50 that contain the string *get*

Most range specifications can be combined with a text buffer specification. During your editing session, you may wish to hold and edit text in buffers other than MAIN. To create and gain access to alternate buffers, include the name of the buffer in a range specification, using the following syntax:

```
=buffer [range]
  or
BUFFER buffer [range]
```

In this syntax, "buffer" stands for the name of the buffer. It can be from 1 to 30 alphanumeric characters, but it must start with an alphabetic character. If you include a range of lines following the buffer name, you specify the range within the named buffer. If you omit the range specification, you specify either the entire named buffer or its first line, depending on context.

The following examples show buffer specifications in use.

Specification	Meaning
=PROG1	The entire contents of the text buffer named PROG1, or (for commands requiring a single-line range specification) its first line.
=INC 'sub1()':'}'	The lines that contain the strings *sub1()* and } in the text buffer named INC, and all lines in between.
=COM ALL 'copy()'	All lines that contain the string *copy()* in the buffer named COM.

13.4.2 Maneuvering in the File

This section describes commands for maneuvering in a buffer containing text; in other words, for changing the location of the current line.

The TYPE command, followed by a range, causes EDT to display the line or lines in the range and resets the current line to the first (or only) line displayed. The word TYPE (abbreviation T) is optional; it need not be entered. For example:

```
*T 1:3
     1          This is the first line of EXAMPLE.TXT
     2          This is the second line of EXAMPLE.TXT
     3          This is the third line of EXAMPLE.TXT
*4#2
     4          This is the fourth line of EXAMPLE.TXT
     5          This is the fifth line of EXAMPLE.TXT
*
```

If you do not include the word TYPE, and if the range specification begins with an alphabetic character (such as WHOLE or REST), you must precede it with a percent sign (%). Otherwise, EDT tries to interpret the range specification as a command. For example:

```
*REST
  ^
Unrecognized command
*%REST
     4          This is the fourth line of EXAMPLE.TXT
     5          This is the fifth line of EXAMPLE.TXT
     6          This is the sixth line of EXAMPLE.TXT
     7          This is the seventh line of EXAMPLE.TXT
*
```

A carriage return in response to the asterisk prompt displays the line following the current line and sets the current line to the displayed line. A series of carriage returns, therefore, displays successive lines and sets the current line to the displayed line each time. This is an easy way to work through a file line by line. For example:

```
*(RET)
     5          This is the fifth line of EXAMPLE.TXT
*(RET)
     6          This is the sixth line of EXAMPLE.TXT
*
```

The FIND command (abbreviation F) locates a specified line without displaying it. It is useful for setting the current line to the top of a large block of text that would be cumbersome to display on the terminal. For example, each of the following commands resets the current line to the top of the MAIN text buffer:

```
*=MAIN

*F =MAIN
```

However, the first command (an implied TYPE command) displays the entire contents of the MAIN text buffer. The second command just sets the current line and displays an asterisk prompt.

If you specify a range that EDT cannot locate, EDT issues a message and does not change the current line setting.

13.4.3 Inserting New Text

The procedure for inserting new text in a buffer already containing text is exactly the same as that for inserting text in an empty buffer (see Section 13.3), except that you can control where the text goes by including a range specification with the INSERT command. The lines you insert are placed in front of the line you specify. If you specify multiple lines, the insert goes in front of the first line in the range. If you omit the range specification, the insert goes in front of the current line.

In the following example, the INSERT command causes EDT to insert text in front of line 5 in the current buffer. Then the range specification (an implied TYPE command) causes EDT to display lines 4 through 6, showing the result of the insertion.

```
*I 5
                First insert line
                Second insert line
                Third insert line
                CTRL/Z ^Z
* 4:6
     4          This is the fourth line of EXAMPLE.TXT
     4.1        First insert line
     4.2        Second insert line
     4.3        Third insert line
     5          This is the fifth line of EXAMPLE.TXT
     6          This is the sixth line of EXAMPLE.TXT
*
```

NOTE

EDT, which inserts text in front of the current line, is different from many other text editors that insert text following the current line.

13.4.4 Deleting and Replacing Text

Use the DELETE command (abbreviation D) to delete a specified range. If you omit the range, the DELETE command deletes the current line. After a delete operation, EDT displays the line following the last line deleted; this is the new current line. For example:

```
*D 4.1#2
2 lines deleted
     4.3          Third insert line
*D
1 line deleted
     5            This is the fifth line of EXAMPLE.TXT
*
```

The /QUERY qualifier to the DELETE command causes EDT to prompt you before deleting each line of the range. The prompt is a question mark (?). You can respond to the prompt in one of four ways:

Y (yes)	Delete this line
N (no)	Do not delete this line
A (all)	Delete all remaining lines in the specified range
Q (quit)	Quit the delete operation

The REPLACE command (abbreviation R) deletes a specified range and allows you to insert lines to replace the deleted lines. You terminate the insertion with a CTRL/Z, just as with the INSERT command.

13.4.5 Moving Text

The COPY and MOVE commands (abbreviations CO and M, respectively) allow you to move one or more lines of text from one place in the buffer to another, or from one buffer to another. The effect of these commands is similar; the only difference is that the COPY command does not delete the text from its original location, whereas the MOVE command does.

The following example illustrates both commands, as well as alternative ways of specifying a range:

```
*%WHOLE
     1          This is the first line of EXAMPLE.TXT
     2          This is the second line of EXAMPLE.TXT
     3          This is the third line of EXAMPLE.TXT
     4          This is the fourth line of EXAMPLE.TXT
     5          This is the fifth line of EXAMPLE.TXT
     6          This is the sixth line of EXAMPLE.TXT
     7          This is the seventh line of EXAMPLE.TXT
*COPY 1:3 TO 'SIXTH'
3 lines copied
*5:6
     5          This is the fifth line of EXAMPLE.TXT
     5.1        This is the first line of EXAMPLE.TXT
     5.2        This is the second line of EXAMPLE.TXT
     5.3        This is the third line of EXAMPLE.TXT
     6          This is the sixth line of EXAMPLE.TXT
*M 5.1#3 TO BEGIN
3 lines moved
*%WH
     0.1        This is the first line of EXAMPLE.TXT
     0.2        This is the second line of EXAMPLE.TXT
     0.3        This is the third line of EXAMPLE.TXT
     1          This is the first line of EXAMPLE.TXT
     2          This is the second line of EXAMPLE.TXT
     3          This is the third line of EXAMPLE.TXT
     4          This is the fourth line of EXAMPLE.TXT
     5          This is the fifth line of EXAMPLE.TXT
     6          This is the sixth line of EXAMPLE.TXT
     7          This is the seventh line of EXAMPLE.TXT
*
```

The /QUERY qualifier to either COPY or MOVE causes EDT to prompt you before copying or moving each line of the range. It operates the same way as the /QUERY qualifier to DELETE (see Section 13.4.4).

13.4.6 Substituting Text

Two commands, SUBSTITUTE and SUBSTITUTE NEXT, substitute one string for another within a line or lines. These are the only line

editing commands that can alter text within a line, as opposed to changing the entire line. The SUBSTITUTE command (abbreviation S) operates on the current line or on a specified range; the SUBSTITUTE NEXT command (abbreviation N) makes a substitution at the next opportunity within the buffer.

The format of the SUBSTITUTE command is:

SUBSTITUTE /string1/string2/[range] [/QUERY]

The command finds string1 and substitutes string2 for it. If you do not specify a range, the substitution takes place in the current line. If you do, the command makes every substitution within the range. The following example illustrates the command first without and then with a range specified:

```
*1
     1            This is the first line of EXAMPLE.TXT
*S /first/1st/
     1            This is the 1st line of EXAMPLE.TXT
1 substitution
*S /of/in/4:6
     4            This is the fourth line in EXAMPLE.TXT
     5            This is the fifth line in EXAMPLE.TXT
     6            This is the sixth line in EXAMPLE.TXT
3 substitutions
*
```

Slashes (/) are not the only characters you can use to delimit string1 and string2. Any nonalphanumeric character will work, as long as the delimiters are matched and do not occur in either string. For example, the following command substitutes the string *a/3* for *a/2* in the current line, using dollar signs ($) as delimiters:

```
*S $a/2$a/3$
    25               size = a/3;
1 substitution
*
```

The /QUERY qualifier to SUBSTITUTE causes EDT to prompt you before making each substitution. It operates the same way as the /QUERY qualifier to DELETE (see Section 13.4.4).

The SUBSTITUTE NEXT command (abbreviation N) substitutes for the next occurrence of string1 that it finds in the buffer. If you specify neither string1 nor string2, the command takes their values from the last SUBSTITUTE command you issued. For example:

```
*N / in/ of/
     4            This is the fourth line of EXAMPLE.TXT
*N
     5            This is the fifth line of EXAMPLE.TXT
*
```

13.4.7 Input from and Output to Files

Two EDT commands, INCLUDE and WRITE, allow you to incorporate text from files and output text to files during your editing session. The INCLUDE command (abbreviation INC) incorporates the contents of a file at a specified location in a text buffer. If you do not want the entire file incorporated in the MAIN text buffer, you can specify an alternate buffer as the range, and then copy the desired portions of the file to their proper places in MAIN. For example:

```
*INC SBRTNES.C =SUBS
*
```

This command creates a buffer called SUBS and fills it with the contents of the file SBRTNES.C from the EDT default directory (that is, the directory of the input file given with the EDIT/EDT command).

The WRITE command (abbreviation WR) creates a file by copying the contents of a specified range in a text buffer. The text is not deleted from the text buffer, and EDT does not terminate following the operation. If you do not specify a range with the write command, EDT outputs the entire contents of the current text buffer. The following example shows the command used with a range:

```
*WR ROUTINE1.C =SUBS 'add:':'return'
_DB1:[PROJECT]ROUTINE1.C;1 45 lines
*
```

This command creates the file ROUTINE1.C from the lines that contain the strings *add:* and *return* in the buffer named SUBS, and all lines in between.

Unless you include a directory in the file specification, WRITE always creates the file in your current default directory. This is true even if the input and output files are in another directory.

13.4.8 Editing a File from Another Directory

You can edit a file that exists in another directory and use the /OUTPUT qualifier to EDIT/EDT to direct the output file to your directory. However, EDT uses the directory of the input file that you specify in the EDIT/EDT command line as its default directory. This default has the following effects:

- EDT attempts to create its journal file in its default directory, that is, the other directory. If you do not have the privilege to do this, EDT issues an error message and terminates. You should instead use the /JOURNAL qualifier to place the journal file in your directory. (See Section 13.6 for a description of the journal file and /JOURNAL.)
- If you issue an INCLUDE command and do not specify a directory, EDT attempts to locate the file in its default directory, that is, the other directory. To specify a file in your own directory, use a directory specification with INCLUDE.

In the following example, a user with the account [WILBUR] edits a file from the account [PROJECT]:

```
$ EDIT/EDT [PROJECT]DATADEF.C -
$_/OUTPUT=[WILBUR] /JOURNAL=[WILBUR]
    .
    .
*INCLUDE [WILBUR]ENTRIES.C
```

The input file for this editing session is [PROJECT]DATADEF.C; the output file is [WILBUR]DATADEF.C. The INCLUDE command incorporates a file from directory [WILBUR]. If the INCLUDE command had not specified a directory, EDT would have looked for the file [PROJECT]ENTRIES.C.

13.5 Character Editing

EDT's character editing mode allows you to perform editing operations at any position in your text instead of line by line. For most applications, especially those requiring extensive detail modification of existing text, character editing is faster and more straightforward than line editing. When you use character editing mode on a video terminal, your screen always contains an accurate picture of the area of the file in which you are working. The terminal's cursor shows exactly where you are at all times.

There are two types of character editing: nokeypad and keypad. Nokeypad character editing works on all terminals, including hard-copy terminals. It requires you to enter short commands through the keyboard and terminate each command with a RET. Keypad character editing works on the VT52 and VT100 video terminals and on terminals that are compatible with them. In keypad editing, you request editor functions by pressing keys on the auxiliary keypad; no RET is required to terminate the command. Anything you type on the keyboard, including carriage returns, is inserted into the file as text.

This section describes only keypad character editing. To learn about nokeypad character editing, read the *VAX-11 EDT Editor Reference Manual.*

The keypads for the VT52 and VT100 (and compatible) terminals are different. Therefore, the following description refers to functions rather than to specific keys. It is a good idea to keep a copy of the appropriate keypad diagram handy while you are learning character editing. Figures 13–1 and 13–2 show the keypad diagrams for the VT52 and VT100, respectively. The numbers or characters in the upper right of each key correspond to the label on the key.

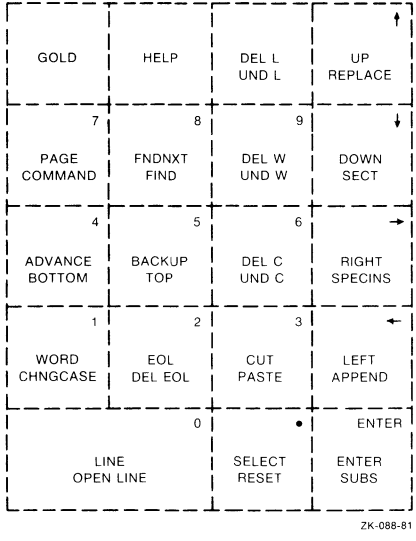


Figure 13-1: VT52 Keypad

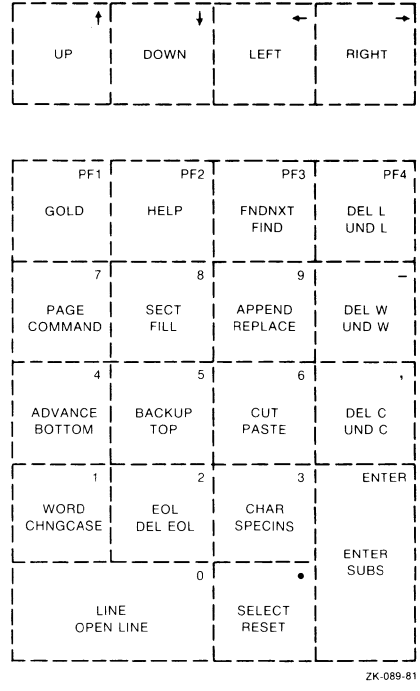


Figure 13-2: VT100 Keypad

Note that most keys perform two functions. To use the upper of the two functions listed, press the key. To use the lower function, first press and release the GOLD key, then press the desired key.

13.5.1 Entering and Exiting from Character Editing Mode

To enter character editing mode from line editing mode, use the CHANGE command (abbreviation C). When you issue the CHANGE command, the screen first goes blank and then fills with text. The cursor is positioned at the current line or the line you specified with the CHANGE command. (If the buffer is empty, the cursor and [EOB] appear at the top of the screen.)

EDT does not display line numbers while in character editing mode, although it does continue to assign them as you insert text.

When you have finished your character editing operations and wish to return to line mode, enter a CTRL/Z. It terminates character editing and causes EDT to display the asterisk prompt. You can then perform line editing operations or end the editing session, as appropriate.

The sections that follow describe some of the character editing operations available to you.

13.5.2 Maneuvering the Cursor

Before performing most character editing operations, you must move the cursor to the location in the file where you wish the operation to take place. There are many ways to move the cursor; experience teaches which is best in a given situation.

The LEFT and RIGHT functions move the cursor one character to the left or right. If the cursor is at the end of a line, the RIGHT function moves it to the beginning of the next line. Conversely, if the cursor is at the beginning of a line, the LEFT function moves it to the end of the previous line.

The UP and DOWN functions move the cursor one line up or down. The column position of the cursor does not change, unless there is no text in the corresponding column above or below. In that case, the cursor moves to the end of the preceding or following line.

The beginning-of-line function, obtained by pressing the BACK SPACE key, moves the cursor to the beginning of the line in which it is positioned. If the cursor is already at the beginning of a line, the function moves it to the beginning of the previous line.

The TOP and BOTTOM functions move the cursor to the beginning and end of the buffer, respectively.

All the remaining cursor movement functions depend in part on the ADVANCE and BACKUP functions. The ADVANCE function causes subsequent cursor movement to occur in the forward direction, that is, toward the end of the buffer. The BACKUP function causes subsequent

cursor movement to occur in the backward direction, toward the beginning of the buffer. When character editing begins, cursor movement is forward, until reversed by the BACKUP function.

The following functions depend on the current direction established by ADVANCE and BACKUP:

- The CHAR function moves the cursor one character.
- The WORD function moves the cursor to the beginning of the next or previous word (the end-of-line character is considered a word).
- The LINE function moves the cursor to the beginning of the next line, if the current direction is forward. If backward, the LINE function moves the cursor to the beginning of the line in which the cursor is positioned, or, if the cursor is at the beginning of a line, to the beginning of the previous line.
- The EOL (for end-of-line) function moves the cursor to the next or previous end-of-line character.
- The SECT (for section) function moves the cursor one 16-line section.
- The PAGE function moves the cursor to the next or previous page mark (by default, a form feed).

All of these cursor movement functions can be combined with a repeat count, which causes the function to be repeated a specified number of times. To enter a repeat count, press the GOLD key, then type in the count on the keyboard (not keypad) number keys, then type in the function to be repeated. As you enter the repeat count, the numbers appear on the screen below the area reserved for text. The numbers disappear as soon as you enter the function.

You can also use FIND and FNDNXT (for find next) to move the cursor to a certain string. To find a string, press the FIND function key. EDT prompts you for a search string. Type the search string without delimiters, and terminate it with either the ADVANCE or BACKUP function to determine the direction of search. EDT moves the cursor to the beginning of the search string. If the search string is not found, EDT issues a message and does not move the cursor.

The FNDNXT function finds the next occurrence of the current search string in the current direction. The current search string is the last string you entered with the FIND function.

Note that you can locate strings that include carriage returns with the FIND function. Simply enter the carriage return as part of the search string. The carriage return does not terminate the search string; you do that with the ADVANCE or BACKUP function. EDT echoes a carriage return in a search string as ^M.

13.5.3 Inserting Text

Once the cursor is positioned, you can insert text in front of it simply by typing the text on the keyboard. No command is required. Whatever you type becomes part of the file. Your insertion appears on the screen as you type it, and the surrounding text moves as necessary.

When you insert text at the beginning or in the middle of a line, the end of the line may disappear off the edge of the screen. The text is not lost. However: if you enter a carriage return in the text you are typing, the text appears on the next line. To avoid this problem, you can use the OPEN LINE function. When the cursor is at the beginning of a line, OPEN LINE provides a blank line above that line, and positions the cursor at the beginning of the blank line.

As you type new text, you may notice errors in surrounding text. You can move the cursor to these errors and correct them at any time, and then move the cursor back and continue to insert text.

13.5.4 Deleting and Undeleting Text

EDT character editing provides several methods of deleting text in units of varying sizes. EDT also maintains three buffers to contain text that has been deleted. The character buffer contains the last character deleted; the word buffer contains the last word deleted; and the line buffer contains the last line deleted. You can insert the contents of each of these three buffers at the cursor position by using the UND C, UND W, and UND L functions, respectively. There is no limit to the time or number of operations between a delete operation and the undelete operation that reinserts the deleted text. Furthermore, you can undelete one unit of text as many times as you wish, and at any locations you wish.

The DEL C (for character) function deletes the character at which the cursor is positioned, and moves the cursor to the next character. The DELETE key on the keyboard deletes the character before the cursor position (the last character typed, if you are inserting text) but does not change the cursor position. Both of these functions move the deleted character into the character buffer, from which it can be retrieved by using the UND C function.

The DEL W (for word) function deletes text from the current cursor position to (but not including) the first character of the next word. The LINE FEED key on the keyboard deletes text from (but not including) the cursor position back to the first character of the current word. Both of these functions move the deleted text into the word buffer, from which it can be retrieved by using the UND W function.

The DEL L (for line) function deletes text from the cursor position through the next end-of-line character. The DEL EOL (for end-of-line) function is similar, except that it does not delete the end-of-line character. Typing CTRL/U deletes from (but not including) the cursor position to

the beginning of the current line. All of these functions move the deleted text into the line buffer, from which it can be retrieved by using the UND L function.

13.5.5 Moving Text

Character editing provides two basic methods of moving text. The first is available through the three undelete functions. You can delete a unit of text from one location, move the cursor to another location, and undelete the text there. However, this method is only effective for units that can be deleted by the various functions described in Section 13.5.4. To move larger or more precise blocks of text, use CUT and PASTE. These two functions allow you to "cut" any amount of contiguous text from one location and "paste" it somewhere else.

The first step is defining the text to be moved. To do this, move the cursor to either the beginning or the end of the text, and enter the SELECT function. Then, move the cursor to the other extremity of the text. In so doing, you create a select range: that is, all the text between the cursor position and the position at which you entered the SELECT function. On VT100 terminals with the advanced video option, EDT highlights the select range with reverse video. If you make a mistake while you are defining the select range, enter the RESET function to cancel the select range currently in effect.

Once you have defined the select range, enter the CUT function. The text within the select range disappears. (EDT moves it into a text buffer named PASTE.) Move the cursor to the position at which the text is desired, and enter the PASTE function. The text appears at the cursor position.

You can paste the cut text in as many locations as required. Specifically, you can paste the text as soon as you cut it, then you can move the cursor and paste the text again. This is in effect a copy operation.

Each CUT operation destroys the previous contents of the PASTE buffer and replaces them with the select range. To add the select range to the contents of the PASTE buffer, use the APPEND function.

The PASTE buffer is an ordinary EDT text buffer. You can edit within it, load it from a file with the INCLUDE command, and create a file from its contents with the WRITE command.

13.6 Protecting and Recovering Text

Three qualifiers to the EDIT/EDT command allow you to protect files against inadvertent modification and to recover editing operations that have been lost. This section discusses them.

The /READ_ONLY qualifier controls whether journaling and the creation of an output file are enabled. (Specifying /READ_ONLY is equivalent to specifying /NOOUTPUT and /NOJOURNAL.) /NOREAD_

ONLY, the default, allows EDT to create an output file and a journal file. Use /READ_ONLY in situations where you want to be sure you do not create a modified file, or for reading a file in a directory where you do not have write privileges.

The /JOURNAL qualifier allows you to disable (/NOJOURNAL) or to specify the name of the journal file that EDT creates to record your editing activity. By default, EDT creates a journal file with the file name of the input file and a file type of JOU. If the editing session ends abnormally, EDT can use the contents of the journal file to re-create the session. If the editing session ends normally (that is, as the result of an EXIT or QUIT command without a /SAVE qualifier), EDT deletes the journal file.

The /RECOVER qualifier causes EDT to use the contents of a journal file to re-create a previous editing session, perhaps one that was lost as the result of an accidental CTRL/Y or system problem. If you specify /RECOVER, EDT locates a file with the same name as the input file and a file type of JOU, then it applies all the editing operations recorded in the journal file to the input file. These operations appear on your terminal as EDT performs them. When EDT has exhausted the contents of the journal file, the activity on the terminal ceases. You can now continue to edit.

Two notes of caution are necessary. First, it is important for the EDIT/EDT command that starts a recovery operation to match exactly the command that started the lost session, including any special start-up command files. The only difference between the two commands should be the /RECOVER qualifier. In particular, the input file must be the same version that you started with at the beginning of the lost session. Second, note that EDT does not necessarily recover your session to the exact point where it was lost. A few keystrokes may be missing.

13.7 EDT Aids for the Programmer

In addition to the general-purpose editing operations discussed thus far, EDT provides some advanced functions that are especially useful for programming. The following sections introduce some of these.

13.7.1 Structured Tabs

Although C is a free-form language, in which excess spaces and tabs have no significance, it is common practice to indent lines to indicate the relationship of statements. It is laborious to enter repeatedly the correct combination of tabs and spaces to achieve the desired indention. EDT solves this problem by providing a system of structured tabs in character editing mode. While you are inserting text, a depression of the tab key inserts the correct combination of tabs and spaces to bring the cursor to the desired column. When you need to begin lines at a

different column, you can increase or decrease the indention level to move the starting column to the left or right by a preset increment.

To use the structured tab feature, follow these steps:

1. While in line editing mode, set the increment between tabs by issuing the SET TAB command with a suitable value. For example:

```
*SET TAB 4
*
```

   At this point, the first TAB on a line (while in character editing mode) positions the cursor at column 5. Subsequent tab stops are at the normal locations.
2. When you want to change the indention level, use CTRL/E or CTRL/D. Each depression of CTRL/E increases the indention by one increment; the first tab stop is n spaces further to the right, where n is the number you gave with the SET TAB command. Pressing CTRL/D decreases the indention level.
3. If you want to set the indention level to correspond to a given column, position the cursor at that column and press CTRL/A. The column must be at an even multiple of n spaces from the left edge of the screen.
4. If you want to change the indention of a block of lines, first define a select range that includes the lines to be shifted. (To define a select range, position the cursor at one end of the block of lines, enter the SELECT function, and then position the cursor at the other end.) Then enter a repeat count (the GOLD key followed by a number typed on the keyboard) to indicate how many units of n spaces the lines should be shifted. A positive repeat count shifts the lines to the right; a negative repeat count shifts the lines to the left. Finally, press CTRL/T.

13.7.2 Special-Purpose Key Definitions

EDT allows you to redefine the functions invoked by all the keys on the auxiliary keypad and many control characters as well. There are two ways to redefine a key's function:

- While in character editing mode, press CTRL/K. EDT prompts you to press the key you wish to define. Once you have pressed the key, EDT prompts you to enter the new function. You can do this either by typing the nokeypad commands that make up the function, or by pressing the keypad keys that correspond to the functions you require. You must follow the function specification with a period. The ENTER function terminates a definition of this type.
- While in line editing mode, issue the DEFINE KEY command. You define the new function to perform as a string of nokeypad character editing commands, followed by a period. The string and period must be enclosed in quotes.

Key redefinition requires a good grasp of nokeypad character editing syntax, as well as a good deal of practice. The EDT help facility (particularly HELP DEFINE KEY and HELP CHANGE SUBCOMMANDS) and the *VAX-11 EDT Editor Reference Manual* are good sources of information. However, this section describes one common application: the redefinition of a key to insert a string of text.

While writing a program, you may find that you are typing the same group of words over and over. For example, you might get tired of typing c = getchar(). In character editing mode, follow this procedure to define a key to insert the string c = getchar():

1. Press CTRL/K. EDT prompts you with:

   ```
   Press the key you wish to define
   ```

2. Select a function that you do not use often, for example, SPECINS. You might also select a control character. Enter the function or control character. EDT then prompts you with:

   ```
   Now enter the definition terminated by ENTER
   ```

3. Type the following:

   ```
   ic = getchar()CTRL/Z .
   ```

   (The period is required syntax.)
4. Press ENTER to terminate the definition procedure.

For the remainder of the editing session, the key that used to invoke the SPECINS function instead inserts the string c = getchar() at the cursor position.

In line editing mode, you can redefine a key by using the DEFINE KEY command. To identify a keypad key in the command, you use a number. You can find out which numbers are assigned to which keys by issuing the command HELP DEFINE KEY VT52 or HELP DEFINE KEY VT100. These commands display the numbers assigned to keypad keys on the respective terminals.

Next, you issue a DEFINE KEY command, specifying the key and the function you wish the key to perform. The following example redefines the SPECINS function (GOLD/3 on a VT100) to insert the string c = getchar():

```
*DEFINE KEY GOLD 3 AS "ic = getchar()^Z."
*
```

The quotes and period are required syntax. The ^Z is *not* a CTRL/Z, but a circumflex followed by a Z. For the remainder of the editing session, GOLD/3 will insert the string c = getchar() at the cursor position.

The examples above represent only a small fraction of the capabilities of key redefinition. With practice, you can create powerful custom functions that can save you a great deal of time. You may want to store these functions in a start-up command file so that you will not have to define them each time you begin an editing session. The next section describes start-up command files.

13.7.3 Start-Up Command Files

When you invoke EDT, it searches your current default directory for a file named EDTINI.EDT. If EDT finds such a file, it executes the line editing commands contained in the file before turning control over to you. This function allows you to customize EDT to suit your needs. Some of the commands that a start-up command file might contain are:

- DEFINE KEY. These commands redefine the function invoked by a keypad key or control character while in character editing mode. (See Section 13.7.2.)
- DEFINE MACRO. These commands associate a name with a sequence of line editing commands stored in a text buffer. You can then invoke the sequence by entering the macro name in response to the line editing asterisk prompt.
- INCLUDE. These commands bring text from a file into a text buffer. You might use them to load macros into a buffer, or to fill a buffer with text that you often use. (See Section 13.4.7.)
- SET. These commands establish EDT operating parameters. Particularly useful are SET TAB, which establishes the increment for structured tabs, and SET MODE CHANGE, which causes EDT to enter directly into character editing mode. (Section 13.7.1 describes the use of structured tabs.)

You can use the /COMMAND qualifier to the EDIT/EDT command to cause EDT to search for a file other than EDTINI.EDT. This means that you can have several start-up command files, each designed for a particular application. You may want to include a command in your login command procedure file to equate a short mnemonic to an EDIT/EDT command that invokes a special start-up command file. For example, if you have the following line in your login command file:

```
$ EDC :== "EDIT/EDT/COMMAND=C.EDT"
```

then the command:

```
$ EDC METRIC.C
```

invokes EDT with the start-up command file C.EDT to edit the file METRIC.C.

Chapter 14

Compiling, Linking, and Executing C Programs

If the VAX-11 C compiler and its associated libraries are installed on your system, you can compile, link, and execute a self-contained C program with the commands shown in the following example:

```
$ DEFINE LNK$LIBRARY SYS$LIBRARY:CRTLIB.OLB (RET)
$ CC ZENO (RET)
$ LINK ZENO (RET)
$ RUN ZENO (RET)
```

where ZENO.C is the program source file.

The term *self-contained* means that:

- A single source file (here, ZENO.C) specifies all the C source text, either explicitly or with **#include** control lines.
- The program is written entirely in C and uses only those external functions that are included in one of the following:
  - — A user-defined object library associated with the logical name LNK$LIBRARY.
  - — A user-defined text library associated with the logical name C$LIBRARY.
  - — The DIGITAL-supplied text library SYS$LIBRARY: CSYSDEF.TLB.
  - — The DIGITAL-supplied object libraries SYS$LIBRARY: VMSRTL.EXE or SYS$LIBRARY:STARLET.OLB.

In this example, the library SYS$LIBRARY:CRTLIB.OLB, which contains the object code for all the functions described in Chapter 6, is made the equivalence name for LNK$LIBRARY. While this logical name assignment is in effect, all references to the VAX-11 C library functions are resolved to SYS$LIBRARY:CRTLIB.OLB by the LINK command.

The logical name equivalent for C$LIBRARY is searched for modules of C text that are named in **#include** control lines but whose libraries either were not found or were not (as in the above example) specified in the CC command. (See Chapter 7 for a description of the forms of **#include** control lines.)

The text library SYS$LIBRARY:CSYSDEF.TLB is supplied by DIGITAL with VAX-11 C and contains the necessary C definition text for calling VAX/VMS system services. This library is searched for any text specified in the "library-lookup" form of **#include** control lines that cannot be located elsewhere.

In practice, many or even most of the programs you write cannot be prepared with the simple forms of the commands shown above, for two reasons.

First, the three commands can accept qualifiers to extend their actions. For example, the CC command as shown above does not cause the compiler to produce a source listing.

More important, you will often want to develop programs that are not self-contained. It is also possible to:

- Construct a program from many separate files of C source text. Each file is compiled separately, producing a separate file of object code. Then, the object files are linked to form the executable image.
- Concatenate several files of C source text with a single CC command. This procedure produces a single object file that is then linked and executed.
- Include, with the **#include** control line, modules (or files) of C source text rather than writing all the source text in your own source files. Text libraries are, of course, used in self-contained programs as well. You can create text libraries to store your own C text. (Text libraries are discussed in Chapter 12.)
- Link the program with libraries of precompiled object code. You can define your own object libraries to store compiled C functions or definitions, or you can use the object libraries supplied with the VAX-11 C compiler. (Object libraries are discussed in Chapter 12.)

14.1 The Compile Command (CC)

The VAX-11 C compile command (CC) performs the following operations:

- Takes the actions specified by **#include**, **#define**, and other control lines encountered in the source file, to modify the contents and/or interpretation of the C source text.
- Checks the validity of C source text and issues warning or error messages for invalid statements.
- Translates the C source text into machine language instructions.

- Groups data and machine language instructions into program sections.
- Writes the program sections into an object module.

Object modules are subsequently combined by the VAX/VMS Linker to form an executable image. When the VAX-11 C compiler creates an object module, it effectively provides the linker with the following information:

- The module name, which is usually the same as the file name (not including the file type). That is, if the file given to the compiler is named ZENO.C, the resulting module name is ZENO.
- A list of all entry points (functions), external variables, and global symbols declared in the module. The linker uses this information when it binds together two or more modules and resolves references to the same name in several modules.
- A summary of the program sections it has created and their attributes, plus the generated machine instruction text and relocation information.
- Traceback information, which is used by the VAX/VMS default condition handler when a run-time error occurs. Traceback information allows the default handler to display a list of the active blocks, in order of activation, as an aid to debugging.
- A symbol table, if you request it. This table lists all internal and external names used in the module, giving definitions of their locations. It is used as a debugging aid with the VAX-11 Symbolic Debugger.

14.1.1 CC Command Format

The syntax of the CC command and its qualifiers is as follows:

```
CC command:
    CC[/qualifier...] file-spec-list

file-spec-list:
    file-spec[/qualifier...]
    file-spec-list , file-spec[/qualifier...]
    file-spec-list + file-spec[/qualifier...]
```

Command Qualifier	Default
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=option]	/DEBUG=TRACEBACK
/[NO]LIST	/NOLIST (interactive mode) /LIST (batch mode)
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OBJECT	/OBJECT
/[NO]OPTIMIZE	/OPTIMIZE
/SHOW[=(option,...)]	/SHOW=(NOINCLUDE,SOURCE, NOSTATISTICS,NOEXPANSION, NOINTERMEDIATE,NOSYMBOLS)
/STANDARD=[NO]PORTABLE	/STANDARD=NOPORTABLE
/[NO]WARNINGS	/WARNINGS

File Qualifier

/LIBRARY

The qualifiers are described in detail in Section 14.1.6.

14.1.2 Specifying Input Files

The file-spec-list specifies one or more files of C source text and, optionally, libraries of C text that are searched for modules named in **#include** control lines.

If a file specification does not contain a file type, the compiler assumes a default file type of C for a source file. If the file specification is qualified with the /LIBRARY qualifier and does not contain a file type, the compiler assumes a default file type of TLB for a text library file.

14.1.3 Compiling Files Into Separate Object Modules

File specifications separated by commas are compiled into separate object modules. By default, each object module is placed in a file of the same name as the source file and with the file type OBJ. For example, the command

```
CC MDDATE,ZENO,METRIC RET
```

creates the object module files MDDATE.OBJ, ZENO.OBJ, and METRIC.OBJ.

14.1.4 Compiling Files Into One Object Module

File specifications separated by plus signs are compiled into a single object module, as if the source files in the list were a single source file. By default, the object module is placed in a file with the same name as the first source file in the list and with the type OBJ. For example, the command

```
CC MDDATE,ZENO+METRIC RET
```

creates the object module files MDDATE.OBJ and ZENO.OBJ, where ZENO.OBJ contains the object code resulting from the source code in files ZENO.C and METRIC.C.

Compiling two or more files this way produces the same results as compiling the same text from a single file. Consider, for example, the following:

```
CC ALPHA + BETA + GAMMA,DELTA RET
```

This command produces two object modules in separate files: ALPHA.OBJ and DELTA.OBJ. The scope of any function definitions or external data definitions in ALPHA.C, BETA.C, and GAMMA.C includes the files that follow the definition in the plus-sign list. If BETA.C contains the external data definition

```
double sum;
```

then sum denotes a double-precision variable in BETA and GAMMA, without further declaration. In ALPHA.C, because it precedes the definition, the variable must be declared with:

```
extern double sum;
```

The same **extern** declaration must be used in DELTA.C, since it is not part of the same object module.

Note that the component files of ALPHA.OBJ need not represent complete C programs (or C statements or expressions) as long as the combination of all the source files results in a valid C program.

Note also that the qualifiers described in Section 14.1.6 affect all files in a plus-sign list. For example, the command

```
CC ALPHA + BETA + GAMMA/LIST RET
```

produces an object file and a listing file of the source text from all three files.

14.1.5 Specifying Library Files

When you specify a library file in a CC command, you must precede the file specification of the library with a plus sign and use the /LIBRARY qualifier. For example:

```
$ CC APPLIC+DATAB/LIBRARY
```

This CC command compiles the source program APPLIC.C and uses the library DATAB.TLB to locate any **#include** module references of the form:

```
#include text-module-name
```

The module name must not be enclosed in quotation marks or angle brackets.

When more than one library is specified in a CC command, the compiler searches the libraries in the specified order each time it processes an **#include** control line that specifies a text module name. For example:

```
$ CC APPLIC+DATAB/LIBRARY+NAMES/LIBRARY+SYMS/LIBRARY
```

When the C compiler processes an **#include** control line (with no delimiters around the name in the line) in the source file APPLIC.C, it searches for modules referenced in the libraries DATAB.TLB, NAMES.TLB, and GLOBALSYMS.TLB, in that order.

In a command that requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

```
$ CC METRIC+DATAB/LIBRARY,APPLIC+DATAB/LIBRARY
```

In this example, the C compiler compiles METRIC.C and APPLIC.C separately and uses the library DATAB.TLB for each compilation.

The order in which the library file specifications appear within a concatenated list of files is irrelevant. For example, the following are equivalent:

```
$ CC ALPHA+MYLIB/LIBRARY+BETA
$ CC ALPHA+BETA+MYLIB/LIBRARY
```

After the compiler has searched all libraries specified in the above command(s), it searches the default user library, if any, and then the default library SYS$LIBRARY:CSYSDEF.TLB.

14.1.6 Command and File Qualifiers

Command qualifiers can be placed either on the CC command itself or on individual file specifications. If placed on a file specification, the qualifier affects only the compilation of the specified file. If placed on the CC command, the qualifier affects all files processed by the command unless it is overridden by a qualifier on an individual file specification. The file qualifier /LIBRARY can be placed only on a file specification, not on the CC command.

The rest of this subsection describes the qualifiers individually.

/CROSS_REFERENCE
/NOCROSS_REFERENCE

/CROSS_REFERENCE directs the compiler to generate, in a listing file, cross-references for variable names. The cross-reference lists each line number in the listing file on which each variable is referenced. By default, the compiler does not generate a cross-reference (/NOCROSS_REFERENCE). If /CROSS_REFERENCE is specified, then /LIST and /SHOW=SYMBOLS are implied; a listing file containing a storage map is generated and named according to the defaults described for listing files (see /LIST).

/DEBUG[=option]
/NODEBUG

/DEBUG requests information to be included in the object module for use by the VAX-11 Symbolic Debugger. (For more information on the debugger, see Chapter 15.) You may select one of the options shown below.

Option	Usage
ALL	Includes symbol table records and traceback records. This is equivalent to /DEBUG with no option.
TRACEBACK	Includes only traceback records. This is the default if the /DEBUG qualifier is not present on the command.
NOTRACEBACK	Does not include traceback records. This option is used to exclude all extraneous information from thoroughly debugged program modules. This option is equivalent to /NODEBUG.
NONE	Does not include any debugging information. This is equivalent to /NODEBUG.

/LIBRARY

/LIBRARY indicates that the associated input file is a library containing modules of C source text. If the file specification does not include a file type, CC assumes the default type TLB. A library specification must be preceded by a plus sign and pertains only to the file specification to which it is appended. For example:

```
$ CC ALPHA,BETA+USERLIB/LIBRARY (RET)
```

Presumably, the file BETA.C contains references (in **#include** control lines) to modules in the text library USERLIB.TLB, and the file ALPHA.C does not. (If ALPHA contains such references, the specification USERLIB/LIBRARY must also be appended to ALPHA.)

For more information on the creation and use of text libraries, see Chapter 12.

/LIST[=file-spec]
/NOLIST

/LIST directs the compiler to produce a listing file. If the CC command is executed in interactive mode, the default is /NOLIST. In batch mode, the default is /LIST.

You automatically get a listing file when you specify /CROSS_REFERENCE, /MACHINE_CODE, or /SHOW with one or more of the options INCLUDE, SYMBOLS, EXPANSION, or INTERMEDIATE.

When /LIST is in effect, the compiler, by default, creates a listing file with the same name as the source file and with the file type LIS. If you include a file specification with the /LIST qualifier, that specification is used for the listing file.

/MACHINE_CODE
/NOMACHINE_CODE

/MACHINE_CODE directs the compiler to list the generated machine code in the listing file. The default is /NOMACHINE_CODE. If you specify /MACHINE_CODE, /LIST is implied.

/OBJECT[=**file-spec**]
/NOOBJECT

/OBJECT directs the compiler to produce an object module and is the default. By default, /OBJECT creates an object module file with the same name as the source file (or the name of the first file in a plus-sign list) and with the file type OBJ. If you include a file specification with /OBJECT, that specification is used instead.

The compiler executes more rapidly if it does not have to produce an object module. Use the /NOOBJECT qualifier when you need only a listing of a program or when you want the compiler to check a file of source text for errors.

/OPTIMIZE
/NOOPTIMIZE

/OPTIMIZE directs the compiler to optimize the generated machine code. For example, the compiler eliminates common subexpressions, removes invariant expressions from loops, collapses arithmetic operations into three-operand instructions, and places local variables in registers. /OPTIMIZE is the default. /NOOPTIMIZE instructs the compiler to perform no optimization.

When /DEBUG is specified, the compiler will not perform those optimizations that would affect debugging.

/SHOW=(option,...)

/SHOW sets or cancels listing options. You can select or cancel any of the options shown below.

Option	Usage
[NO]INCLUDE	Prints/does not print the contents of **#include** files and modules in the program listing. NOINCLUDE is the default.
[NO]SOURCE	Prints/does not print the source program statements in the program listing. SOURCE is the default.
[NO]STATISTICS	Prints/does not print compiler performance statistics in the program listing. NOSTATISTICS is the default.

[NO]SYMBOLS	Prints/does not print the symbol table of the compiled program in the program listing. The symbol table includes a list of all functions, the sizes and attributes of all variables referenced in the program, and a program section summary and function definition map. NOSYMBOLS is the default unless /CROSS_REFERENCE is used.
[NO]EXPANSION	Prints/does not print final macro expansions in the program listing. NOEXPANSION is the default.
[NO]INTERMEDIATE	Prints/does not print all intermediate and also the final macro expansions in the program listing. NOINTERMEDIATE is the default.

/STANDARD=PORTABLE
/STANDARD=NOPORTABLE

/STANDARD=PORTABLE directs the compiler to flag certain VAX-11 C language extensions and VAX-11 C relaxations of conventional C language constructs and rules. For example, pointer/integer interchangeability is subject to more stringent tests when /STANDARD=PORTABLE is specified. In summary, /STANDARD=PORTABLE causes the compiler to issue warning messages against C usage that may not be portable between VAX-11 C and other implementations. The default is /STANDARD=NOPORTABLE.

/WARNINGS
/NOWARNINGS

/NOWARNINGS tells the compiler not to display warning (severity W) messages on the terminal or in the listing file (if any). You may find this qualifier useful when you are compiling programs that you know contain statements that cause warnings. The default is /WARNINGS.

14.1.7 Compiler Diagnostic Messages and Error Conditions

One of the functions of the C compiler is to identify syntax errors and violations of language rules in the source program. If it locates any errors, the compiler writes messages to your default output devices (SYS$OUTPUT and SYS$ERROR). Thus, if you enter the CC command interactively, the messages are displayed on your terminal. If the CC command is executed in a batch job, the messages appear in the batch job log file.

If the compiler creates a listing file, it also writes the messages to the listing. Each message in the listing follows the statement that caused the error.

When it appears on the terminal, a message from the compiler has the format:

```
%CC-s-ident, message-text
                    Listing line number m
                    At line number n in name
```

The parts of this message are described below.

%CC
The facility, or program, name of the VAX-11 C compiler. This portion indicates that the message is being issued by VAX-11 C.

s
The severity of the error, represented as follows:

F Fatal error. The compiler stops executing when a fatal error occurs and does not produce an object module. You must correct the error before you can compile the program.

E Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.

W Warning. The compiler produces an object module. It attempts to correct the error in the statement, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.

I Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. No action is necessary on your part.

ident
The message identification: a descriptive abbreviation (mnemonic) of the message text.

message-text
The compiler's message. In many cases, it consists of more than one line of output. A message generally provides you with enough information to determine the cause of the error so that you can correct it.

listing line number *m*
The integer *m* gives the number of the line in the listing file where the error occurs. This information is given when /LIST is specified or implied.

At line number *n* **in** *name*
The integer *n* gives the number of the line where the error occurs. The number is relative to the top of the file or text library module specified by *name*. The **#line** control line can be used to change the line number and name that appear in the message.

The messages produced by the VAX-11 C compiler are listed in Appendix C. The format of an error message in a program listing is shown in Appendix D.

Both the CC command and the DCL command SET MESSAGE give you control over the display of messages. The CC qualifier /NOWARNINGS, discussed previously, suppresses warning messages generated by the compiler. SET MESSAGE lets you decide whether messages will be displayed in their entirety or in a shortened form. For example, if you do not want to see the %CC-s-ident part of messages, you can enter the command:

```
$ SET MESSAGE/NOFACILITY/NOSEVERITY/NOIDENTIFICATION
```

This command cancels the facility, severity, and identification portion of all messages. It remains in effect for all commands you subsequently enter, until you reissue the SET MESSAGE command or log off the system.

14.2 The LINKER Command (LINK)

This discussion of the linker is confined to areas of particular interest to VAX-11 C programmers. For additional information on linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VAX-11 Linker Reference Manual.*

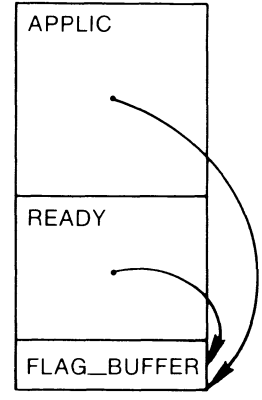
The linker prepares an executable image from object modules. The primary operations of the linker are the allocation of virtual memory within the executable image, the resolution of symbolic references among modules being linked, and the assignment of values to relocatable global symbols. The linker allocates storage for user-defined values, variables, and functions in program sections, whose locations, names, and characteristics depend on the storage class of the variable or other name. The linker also allocates storage for the executable code in the module. The allocation of virtual memory is discussed in Chapter 10.

When two or more object modules are linked, the linker resolves references in one module to variables and other names defined in the modules already linked. This means that the program sections used by the linker allow the modules to use each others' variables and functions. For instance, two C functions can refer to the same **extern** variable because each **extern** variable resides in its own program section. The linker simply resolves an **extern** name to a program section of the same name. Figure 14-1 illustrates the linking of two object modules, in the files APPLIC.OBJ and READY.OBJ, which use the same **extern** variable.

```
$ LINK APPLIC,READY
    unsigned flag_buffer=0x0000;
    applic()
    {

        int val;
          .
          .
        ready(val);
    }
    - - - - - - - - - - - - - - - - - - - - -
    ready(v)
    int v;
    {

        extern unsigned flag_buffer;
        .
        .
    }
```



The linker arranges object modules and the storage allocated to external variables. In the modules APPLIC and READY, all references to the external variable FLAG_BUFFER are resolved to the same virtual storage location.

ZK-085-81

Figure 14–1: Linking Object Modules

Because of the linker's ability to resolve references, modules written in different languages can pass information to each other by referring to the same program section. Chapter 10 includes examples that show variables in C functions sharing program sections with FORTRAN common blocks, PL/I external variables, and MACRO program sections.

The linker must also be used for self-contained programs — those composed of only one object module — because most object modules generated by the compiler contain calls and references to VAX-11 C runtime procedures. The linker automatically locates these procedures in the default system object module libraries. These libraries are described in more detail in Chapter 12.

14.2.1 Format of the LINK Command

The format of the LINK command is:

```
LINK[/qualifier...]  file-spec[/qualifier...],...
```

file-spec,...
The file specifications denote one or more files containing object modules to be linked and, optionally, libraries containing modules.

You can separate the file specifications with commas or plus signs. In either case, all the specified files are used to create a single executable image.

If the file specification does not contain a file type and is not qualified by /LIBRARY, /INCLUDE, or /OPTIONS, the linker assumes a default file type of OBJ.

/qualifier...
The list of qualifiers may include one or more LINK command qualifiers. The /LIBRARY, /INCLUDE, and /OPTIONS qualifiers can only be specified following the specification of an input file. All other qualifiers listed can be specified following either the LINK command or any input file specification.

Table 14–1 summarizes the categories of LINK command qualifiers.

14.2.2 Linker Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any error or fatal conditions occur, the linker will not produce an image file. Since the messages produced by the linker are descriptive, you do not normally need additional information to determine the meaning of a specific error.

Some of the more common errors that occur during linking are as follows:

- The module being linked generated warnings during compilation. You can often link such modules, but you should verify that the modules will produce the results you expect.
- The modules being linked define more than one transfer address. The linker warns you if more than one function in the C program is identified as the main function. The image file created by the linker in this case can be run. The entry point to which control is transferred is the first one found by the linker.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names from the LINK command, and the linker cannot locate the definition for a specified global symbol reference. Such images will not usually execute properly. You can often correct such errors by reentering the command string and specifying the correct modules or libraries.

Table 14-1: LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output files and define a file specification.	/EXECUTABLE[=*file-spec*]	/EXECUTABLE=*name*.EXE, where *name* is the name of the first input file.
	/SHAREABLE[=*file-spec*]	/NOSHAREABLE
	/SYMBOL_TABLE[=*file-spec*]	/NOSYMBOL_TABLE
Request and specify the contents of a memory allocation listing.	/BRIEF	
	/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
	/FULL	
	/[NO]MAP	/NOMAP (interactive) /MAP=*name*.MAP (batch) where *name* is the name of the first input file.
Specify the amount of debugging information.	/[NO]DEBUG	/NODEBUG
	/[NO]TRACEBACK	/TRACEBACK
Indicate that input files are libraries and specifically include certain modules.	/INCLUDE=(*module-name*....)	
	/LIBRARY	
	/SELECTIVE_SEARCH	
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB	/SYSLIB
	/[NO]SYSSHR	/SYSSHR
	/[NO]USERLIBRARY[=*table*]	/USERLIBRARY=ALL
Indicate that an input file is a linker input file.	/OPTIONS	

If an error indicates that a module with a name in the format C$__*name* or C$$__*name* cannot be located, you may not be linking the program with the correct VAX-11 C run-time library. The LINK command must specify the library SYS$LIBRARY:CRTLIB.OLB, or that library must be established as the equivalent of the logical name LNK$LIBRARY.

14.2.3 Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify files containing individual object modules created by a compiler. The linker assumes that any unqualified file specification is an object module.
- Specify one or more object module libraries to be searched to resolve references to external procedures and variables. These libraries are searched for all references that are not resolved among the modules specifically included in the compilation. You must qualify the file specification of the library with the /LIBRARY qualifier.
- Specify explicit modules from an object module library to be included in the image. You must qualify the file specification of the library with the /INCLUDE qualifier and specify the names of the object modules to be included.
- Specify an options file containing additional file specifications and special linker options. You must qualify the file specification of an options file with the /OPTIONS qualifier.

The linker uses these default file types for input files:

File	File Type
Object module	OBJ
Library	OLB
Options file	OPT

You specify object modules and libraries in a command as follows:

```
$ LINK METRIC,-
$_FORMATS/INCLUDE=(PRINTLINE,TERMLINE),-
$_[PROJECT.OBJLIB]MATHLIB/LIBRARY
```

This LINK command links the object module METRIC.OBJ with the modules PRINTLINE and TERMLINE from the library FORMATS.OLB. Any references to external procedures and variables that are not defined in any of these three modules will cause the linker to search the library MATHLIB.OLB in the directory [PROJECT.OBJLIB] before it searches the system libraries.

The format and content of a linker options file are described in detail in the *VAX-11 Linker Reference Manual*. You may wish to use an options file if you have a very long list of input files to be specified, if you want to link a module with a shareable image file, or if you want to request special linker options.

When you specify more than one input file for the LINK command, the linker combines individual object files or library modules in the order in which they are listed.

When you specify libraries as input for the linker, you can specify as many as you wish; there is no practical limit. More than one library can contain a definition for the same module name. The linker uses the following conventions to search libraries specified in the command string:

- A library is searched only for definitions that are unresolved in the previous input files specified.
- If more than one library is specified for an object module, the libraries are searched in the order in which they are specified.

For example:

```
$ LINK METRIC,DEFLIB/LIBRARY,APPLIC
```

The library DEFLIB will be searched only for unresolved references in the object module METRIC. It is not searched to resolve references in the object module APPLIC. However, this command can also be entered as follows:

```
$ LINK METRIC,APPLIC,DEFLIB/LIBRARY
```

In this case, DEFLIB.OLB is searched for all references that are not resolved between METRIC and APPLIC. After the linker has searched all libraries specified in the command, it searches default user libraries, if any, and then the default system libraries.

When one or more logical names exist for default user libraries, the linker uses the following search order to resolve references:

1. The process, then the group, and then the system logical name tables are searched for the name LNK$LIBRARY. If the logical name exists in any of these tables and if it contains the desired reference, the search ends.
2. The process, then the group, and then the system logical name tables are searched for the names LNK$LIBRARY_1 through LNK$LIBRARY_999. If the logical name exists in any of these tables, and if it contains the desired reference, the search ends.

This search sequence is repeated for each reference that remains unresolved.

The search order can be modified for a particular link operation. To override the search of a library, you can do one of the following:

- Delete the logical name of the library you do not want searched. For example:

```
$ DEASSIGN LNK$LIBRARY
```

  The DEASSIGN command deletes the logical name table entry LNK$LIBRARY.

- Specify /USERLIBRARY or /NOUSERLIBRARY on the LINK command. These qualifiers let you specify the PROCESS, GROUP, and SYSTEM keyword options to explicitly control which logical name tables are to be searched for default user libraries. For example:

```
$ LINK/USERLIBRARY=GROUP input-file,,,,
```

  When it executes this command, the linker searches only the GROUP logical name table. Specify /NOUSERLIBRARY to exclude all default user libraries in the search.

For complete details on defining and using default user libraries, see the *VAX-11 Linker Reference Manual.*

14.2.4 Linker Output Files

When you enter the LINK command interactively with no qualifiers, the linker creates only an executable image file. By default, it has the same file name as the first or only object module specified, and it has a file type of EXE. For example:

```
$ LINK A,B,C
```

This LINK command links the object modules in the files A.OBJ, B.OBJ, and C.OBJ, and it creates the image file A.EXE.

In a batch job, the linker creates both an executable image file and a storage map file by default. The default file type for map files is MAP.

Some of the rules for naming input and output files are shown in Table 14–2. These rules also apply to the specification of names with the /MAP qualifier. To specify an alternate name for a map file or image file, or to specify an alternate output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. Some examples are:

Command	Output File(s)
`$ LINK METRIC/MAP=TEST`	METRIC.EXE (by default) TEST.MAP

`$ LINK METRIC/EXE-`
`$_=[PROJECT.EXE]-`
`$_/MAP=[PROJECT.MAP]`
	[PROJECT.EXE]METRIC.EXE
	[PROJECT.MAP]METRIC.MAP

`$ LINK METRIC/MAP=LP:`	METRIC.EXE (by default) line printer listing of the map file

In the third example, the map file is not saved on disk after it is printed.

Table 14-2: Specifying Input and Output Files for the Linker

Rule	Example	Output File(s)
If you do not specify the /EXECUTABLE qualifier, the linker gives the image file the same name as the first input file.	`$ LINK METRIC`	METRIC.EXE
If you specify /EXECUTABLE following the name of an input file, the linker uses that file's name to name the output file.	`$ LINK METRIC,-` `$_APPLIC/EXECUTABLE`	APPLIC.EXE
If you give a file specification with the /EXECUTABLE qualifier, the linker uses that file specification.	`$ LINK/EXECUTABLE-` `$_=TEST METRIC,APPLIC`	TEST.EXE

When you specify a device and/or directory for a file specification, that device and/or directory becomes a temporary default for the remaining input and output files.	`$ LINK METRIC,-`
	`$_[PROJECT]-`
	`$_MATHLIB/LIBRARY,-`
	`$_FORMATS/EXECUTABLE`
		[PROJECT]FORMATS.EXE

14.2.5 Specifying Map File Qualifiers

The map file, also called a memory allocation listing or storage map, describes how the linker has arranged the object modules and their contents in the image file. The map file also lists the virtual memory addresses that the linker has assigned to procedure entry points.

When you specify the /MAP qualifier, or when a map is produced by default in a batch job, the /BRIEF and /FULL qualifiers define the information included in the file, overriding the default content. The types of maps and the qualifiers you use to request them are:

- Brief — specify /MAP/BRIEF
- Default — specify /MAP
- Full — specify /MAP/FULL

The contents of these maps are summarized in Table 14–3. For information on how the VAX–11 describes object modules to the linker and arranges program data according to their attributes, see Chapter 10.

Table 14–3: Contents of a Map File

FULL		
Summary of image characteristics	List of global symbols by name	List of global symbols by value
Names of all modules in the image	List of user-defined program sections	Summary of characteristics of each image and program section in the image
Linker performance statistics		
BRIEF		
DEFAULT		

14.2.6 Specifying Debugging Qualifiers

You can specify either the /DEBUG or /TRACEBACK qualifiers when you link an image. The qualifiers control the amount of debugging information that is available to the VAX–11 Symbolic Debugger and to the run-time error-reporting mechanism.

By default, the linker includes traceback information, which causes the run-time system to list all of the procedure invocations active at the time of a fatal run-time error. If you specify the /NOTRACEBACK qualifier, that information will not be available.

Regardless of whether you specified /DEBUG to the VAX-11 C compiler, you can specify /DEBUG when you link the object module. This qualifier requests that the object modules containing the debugger program be linked to your object modules. When you execute the program, the debugger initially takes control. The steps required to run a program under the control of the debugger and the symbolic debugging capabilities available for C programmers are described in Chapter 15.

14.3 Executing Programs (RUN)

This section describes the considerations involved in executing C programs on the VAX/VMS operating system. For further information on any of the DCL commands or topics presented here, see the *VAX/VMS Command Language User's Guide*.

14.3.1 Image Execution with RUN

You execute a VAX-11 program with the RUN command. The RUN command assumes by default that the file type of a program image is EXE. For example, the command

```
$ RUN METRIC
```

locates the file METRIC.EXE in the current default directory. Control then passes to the main function in the C program. If no function in the program is identified as the main function, then initial control passes to the first, or only, module that was linked into the image.

14.3.2 Command-Line Arguments

The main function in a VAX-11 C program can accept arguments from the command line that invokes it. The synopsis for a main function is as follows:

```
int main(argc,argv,envp)
int argc;
char *argv[ ],*envp[ ];
```

In this synopsis, argc is the count of arguments present in the command line that invoked the program, and argv is a character-string array of the arguments. envp is the environment array. It contains process information, such as the user name and controlling terminal. It has no bearing on passing command-line arguments. Its primary use in C programs is during **exec** and **getenv** function calls. See Chapter 6 for more details on the use of the envp argument.

In the main function definition, the parameters are optional; you can define main in any of the following ways:

```
main()
main(argc)
main(argc,argv)
main(argc,argv,envp)
```

However, you can access only the parameters that you define.

To pass arguments to the main function, you must install the program as a DCL foreign command. When a program is installed and run as a foreign command, argc is always greater than or equal to 1, and argv[0] always contains the name of the image file. Example 14–1 shows a program called COMMARG.C, which displays the command-line arguments that were used to invoke it.

```
#include stdio

/* ECHO COMMAND LINE ARGUMENTS */
main(argc,argv)
int argc;
char *argv[];
{
    int i;

    /* argv[0] IS THE PROGRAM NAME */
    printf("program: %s\n",argv[0]);

    for (i=1; i<argc; i++)
        printf("argument %d: %s\n",i,argv[i]);
}
```

Example 14–1: Echo Program Using Command-Line Arguments

The program is then compiled and linked normally:

```
$ CC COMMARG RET
$ DEF LNK$LIBRARY SYS$LIBRARY:CRTLIB.OLB RET
$ LINK COMMARG RET
```

The procedure for installing a foreign command is described fully in the *VAX/VMS Command Language User's Guide*. Briefly, the procedure uses a DCL assignment statement to assign the name of the image file to a symbol that is subsequently used to invoke the image. For example:

```
$ ECHO :== $ WRK$:COMMARG.EXE RET
```

Here, ECHO is installed as a foreign command that invokes the image in COMMARG.EXE. The definition of ECHO must begin with a dollar sign ($) and include a device name, as shown.

A sample run of the ECHO command follows:

```
$ ECHO Now is "the Time" (RET)
Program: db7:[zeno.src]commarg.exe;1
argument 1: now
argument 2: is
argument 3: the Time
```

The command line you enter is subject to the usual DCL rule of conversion to uppercase for most arguments. VAX-11 C internally parses and modifies the DCL-modified command line to make the command line more compatible with UNIX-developed programs.

All alphabetic arguments in the command line are delimited by spaces or tabs. Arguments that have embedded spaces or tabs must be enclosed in quotation marks ("). Uppercase characters in arguments are converted to lowercase, but arguments within quotation marks are left unchanged.

14.3.3 Image Exit

When the main function executes a **return** statement or reaches the end of its outer block, the image is terminated. In the context of the VAX/VMS operating system, the termination of an image (image exit) causes the system to perform a variety of clean-up operations during which open files are closed, system resources are freed, and so on.

Image exit also occurs as a result of run-time errors or any of the following:

- The execution of the image was interrupted by (CTRL/Y), followed either by a command that executes another image or by the DCL command EXIT. (Note that the DCL command STOP does not cause image exit.)
- A function in the program called the SYS$EXIT system service.
- A process in the system called the SYS$FORCEX system service, forcing the exit of the current process.

For complete details on the actions VAX/VMS takes when an image exits, and for an explanation of the SYS$EXIT and SYS$FORCEX system services, see the *VAX/VMS System Services Reference Manual.*

On image exit, the current contents of general register 0 are delivered to the VAX/VMS command interpreter (DCL) as a status value. To perform properly on image exit, the function must be the first function encountered by the linker, or it must be named main or be defined with the **main-program** option. Otherwise, the function must include one of the following to return a VAX/VMS error code:

- A **return** statement
- A call to **__exit**
- A call to **exit**

14.3.4 Run-Time Errors

When an error occurs during the execution of a program, the program is terminated and one or more messages are displayed by the VAX/VMS condition handler on the current SYS$ERROR device.

A message is followed by a traceback. For each module in the image that has traceback information, the condition handler lists the modules that were active when the error occurred, showing the sequence in which the modules were called.

For example, if an integer divide-by-zero condition occurs, a run-time message like the following appears:

```
%C-F-ERROR, C error condition
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero
    at PC=00000FC3, PSL=03C00002
```

This message is followed by a traceback message similar to the following:

```
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name     line    relative PC    absolute PC
MAIN            MAIN                8       00000007       00000FC3
C$MAIN          C$MAIN           1408       000002F7       00000B17
```

The information in the traceback message is as follows:

module name
The names of image modules that were active when the error occurred. (For errors originating in the C source code, the module names are those created by the CC command or **#module** control line.)

The first module name is that of the module in which the error occurred. Each subsequent line gives the name of the caller of the module named on the previous line. In this example, the modules are MAIN and C$MAIN; C$MAIN called MAIN.

routine name
The name of the function in the calling sequence.

line
The source program (compiler-generated) line number of the statement in which the error occurred, or at which the call or reference to the next procedure was made. Line numbers in these messages match those in the listing file.

relative PC
The value of the PC (program counter). This value represents the location in the program image at which the error occurred or at which a procedure was called. The location is relative to the virtual memory address that the linker assigned to the code program section of the module indicated by module name.

absolute PC
The value of the PC in absolute terms, that is, the actual address in virtual memory representing the location at which the error occurred.

Traceback information is available at run time only for modules compiled and linked with the traceback option in effect. The traceback option is in effect by default for both the CC and LINK commands. You may use the CC command qualifier /NODEBUG and the LINK command qualifier /NOTRACEBACK to exclude traceback information. However, traceback information should be excluded only from thoroughly debugged program modules.

14.3.5 Interrupting a Program

When you execute the RUN command interactively, you cannot execute any other program images or DCL commands until the current image exits. However, if your program is not performing as expected — if, for instance, you believe your program is in an endless loop — you can interrupt it with CTRL/Y. For example, the sequence

```
$ RUN APPLIC
^Y
$
```

interrupts the program APPLIC. After you have interrupted a program, you can terminate it by entering a DCL command that executes another image, or by entering the DCL command EXIT.

Following a CTRL/Y interruption, you can also force an entry to the debugger by entering the DEBUG command. The debugger is described in Chapter 15.

Some other DCL commands have no direct effect on the image. You can enter any of the following commands and then resume the execution of the image with the DCL command CONTINUE:

=	INQUIRE
ALLOCATE	ON
ASSIGN	OPEN
ATTACH	READ
CLOSE	SET CONTROL_Y
DEALLOCATE	SET DEFAULT
DEASSIGN	SET ON
DEBUG	SET PROTECTION/DEFAULT
DECK	SET VERIFY
DEFINE	SET UIC
DELETE/SYMBOL	SHOW DAYTIME
DEPOSIT	SHOW DEFAULT
EOD	SHOW PROTECTION
EXAMINE	SHOW QUOTA
GOTO	SHOW STATUS
IF	SHOW SYMBOL

SHOW TIME	STOP
SHOW TRANSLATION	WAIT
SPAWN	WRITE

14.3.6 Returning Values to the Command Interpreter

The main function in a VAX-11 C program can use the **return** statement to return a status value to the command interpreter. When any program or command is executed under the control of the DCL command interpreter, general register 0 (R0) indicates the completion status. The command interpreter has a special routine that uses the value of R0 to print or display a message on completion of a program.

When the returned value is a numeric value expressible in 32 bits or less, it is placed in R0. Every possible message that can be issued by a system program, command, or component has a unique 32-bit numeric value associated with it. By using this value, the command interpreter locates the message in a central system message file or a user-defined message file.

Note that if you write a main function that returns arbitrary values, the values may be detected by the command interpreter and used to display messages that you would not expect. On the other hand, you can take advantage of this convention and use the **return** statement to exit from a program with a specific status. For example:

```
#include ssdef

main()
{
     .
     .
     return SS$_NORMAL;
}
```

In this example, the **#include** module ssdef defines linker-resolved symbols for standard VAX/VMS status return values.

Under the following conditions, the command interpreter does not display messages on completion of a program:

- A **return** statement specifies the value SS$__NORMAL, denoting normal return status.
- The main function does not return a value. If the main function has no **return** statement, or if its **return** statement specifies no return value, the value SS$__NORMAL is always returned and no message is displayed.
- The status return value suppresses the printing of status messages. (See Chapter 9.)

Chapter 15

Debugging VAX-11 C Programs

The VAX-11 Symbolic Debugger helps you detect logic and programming errors. Specifically, it lets you control the execution of your program so you can monitor specific locations, change the contents of locations, check the program flow, and otherwise locate and correct errors as they occur. This chapter describes those areas of debugging that are specific to VAX-11 C. For a detailed description of the VAX-11 Symbolic Debugger, refer to the *VAX-11 Symbolic Debugger Reference Manual.*

The VAX-11 Symbolic Debugger has many helpful features, among which are the following:

- It is interactive. You control your program and interact with the debugger from your terminal.
- It understands VAX-11 C scalar variable names and their data types. Thus, when you want to look at the contents of a variable, or change the value of a variable, the debugger will convert your ASCII text input to the data type of the variable.
- It understands other programming languages, such as FORTRAN and COBOL. Thus, if your programs consist of procedures written in different languages, you can change from one language to another during the course of a debugging session.

For this version of the VAX-11 C compiler, not all functions of the VAX-11 Symbolic Debugger are supported. This chapter describes the extent of support as it presently exists.

15.1 Using the VAX-11 Debugger

This section gives brief examples that show how to invoke and use the debugger with a VAX-11 C program.

15.1.1 Beginning and Ending a Debugging Session

To execute a VAX-11 C program with the debugger, you must first compile and link the program with the /DEBUG qualifier, as in the following example:

```
$ CC/DEBUG METRIC
$ LINK/DEBUG METRIC
```

The /DEBUG qualifier in the CC command requests the compiler to write symbol table records into the object module. These records permit you to examine and modify variables by name during the debugging session.

The /DEBUG qualifier in the LINK command requests the linker to include the debugger routines, global symbols, and traceback information in the executable image. To include only traceback information, specify /TRACEBACK (which is the default for all LINK commands).

To obtain a program listing and a storage map listing of the functions being debugged, compile the function(s) with the /SHOW=SYMBOLS qualifier added to the /DEBUG qualifier. For example:

```
$ CC/DEBUG/SHOW=SYMBOLS METRIC
```

The /SHOW qualifier can request a listing of **#include** files that are part of your program. To list the storage map and **#include** files (with their statement line numbers), specify:

```
$ CC/DEBUG/SHOW=(SYMBOLS,INCLUDE) METRIC
```

In addition, if you use **#define** macros, you can compile your modules with the /SHOW=EXPANSION qualifier. To list the compiler map, **#include** files, and **#define** macro expansion, specify:

```
$ CC/DEBUG/LIST/SHOW=(SYMBOLS,INCLUDE,EXPANSION) METRIC
```

When you execute an image compiled and linked with the debugger, initial control goes to the debugger, which identifies itself as follows:

```
$ RUN METRIC
                VAX-11 DEBUG Version 'x.xx'
%DEBUG-I-INITIAL, language is BASIC, module set to 'CONVERT'
DBG>
```

For this version of the VAX-11 C debugging support, the language is set to BASIC. The module name displayed in the debugger's message is the name of the object module containing the main function. It is not necessarily the same as the name of the image file. This message indicates that the name of the main function in the image file METRIC is CONVERT.

The DBG> prompt indicates that the debugger is now ready to process your commands. You respond to the prompt with one of the commands recognized by the debugger. (See Table 15–1.)

To terminate the debugging session, use the EXIT command:

```
DBG>EXIT
```

When your program has been thoroughly debugged, you can recompile and relink it without the /DEBUG qualifier. Or, you can run it with the /NODEBUG qualifier. For example:

```
$ RUN/NODEBUG METRIC
```

However, the modules required by the debugger occupy space within a program image file, so recompiling is usually preferable.

15.1.2 The DEBUG Command

When a program that has been linked with /DEBUG is executing, you can interrupt it with CTRL/Y at any time and invoke the debugger by entering the DEBUG command. For example, if you think a program may be looping, or if you see erroneous output, you can interrupt it as follows:

```
$ RUN COMPUTE
^Y

$ DEBUG
DBG>
```

When you press CTRL/Y, the command interpreter displays its dollar sign ($) prompt, and you can enter the DEBUG command. The DBG> prompt indicates that the debugger has control.

If the program was compiled and linked with the /DEBUG qualifier, you have access to full symbolic debugging; you can reference program variables, line numbers, and entry-point names. If the program was not compiled with the /DEBUG qualifier, you have access to limited symbolic debugging; you can reference only entry-point names.

15.1.3 Effects of Optimization on Debugging

When you compile a VAX-11 C program, the resulting object code is optimized; that is, the compiler has used techniques to make the program run faster. For example, the compiler puts automatic scalar variables in registers, removes invariant expressions from loops, and so on.

You do not need to disable any compiler optimizations in order to debug a VAX-11 C program. By default, the compiler does not perform any optimization that would adversely affect debugging when /DEBUG is specified.

15.2 Debugger Command Syntax and Summary

You enter commands to the debugger in much the same way that you enter DCL commands. The debugger commands have the format:

```
cmd [keyword] [/qualifier] [param ...]   !comment
```

cmd
Is a command verb (for example, SET, CANCEL) that indicates the general function to be performed.

keyword
Gives the specific function to be performed by the command (for example, CANCEL MODULE, SET SCOPE, SHOW LANGUAGE).

/qualifier
Modifies the effect of the command.

param
Qualifies the function in some way, such as specifying a range of locations to be monitored.

comment
Is any text message. The debugger ignores all text after the exclamation mark.

You can enter more than one command on a command line by separating the commands with semicolons (;).

You can continue a command on a new line by ending the line with a hyphen (-); the debugger will then prompt for the rest of the command with an underscore (_).

Table 15–1 summarizes the debugger commands. The boldface letters indicate the minimum abbreviation you must type in order for the debugger to recognize the command name, qualifier, or parameter.

You can obtain information about a debugging command with the debugger's HELP command.

Table 15-1: Summary of Debug Commands

@file-spec

Reads debugger commands from the specified command procedure file.

CALL entry-name [(argument,...)]

Invokes a specified function and optionally passes arguments to it.

CANCEL ALL

Cancels all breakpoints, tracepoints, and watchpoints, and restores the mode and scope to their original values.

CANCEL BREAK { /ALL | %LINE line-number | entry-name | symbolic-reference | nonsymbolic-address }

Cancels all breakpoints or a specified breakpoint.

CANCEL EXCEPTION BREAK

Cancels the effect of SET EXCEPTION BREAK and restores the debugger's default method for handling exceptions.

CANCEL MODE

Restores the radix and display modes to their defaults for VAX-11 C debugging, which are decimal and symbolic.

CANCEL MODULE { /ALL | module,... }

Deletes all modules from the run-time symbol table, or deletes one or more modules from the symbol table.

CANCEL SCOPE

Resets the scope to that containing the current program counter.

CANCEL TRACE { %LINE line-number | entry-name | symbolic-reference | nonsymbolic-address | /ALL | /BRANCH | /CALL }

Cancels a specified tracepoint or all tracepoints.

Table 15–1: (Cont.) Summary of Debug Commands

CANCEL TYPE/OVERRIDE

Restores the debugger's default interpretation of variables: the variables' declared data types and sizes.

CANCEL WATCH { /ALL | variable-reference | symbolic-reference | nonsymbolic-address }

Cancels all watchpoints or cancels a watchpoint on a specified location or variable.

DEFINE symbol = expression ,...

Creates one or more symbols whose values are equated to program locations or to numeric expressions.

DEPOSIT location = data [,data,...]

[/ASCII:n | /BYTE | /INSTRUCTION | /LONG | /WORD] [/DECIMAL | /HEXADECIMAL | /OCTAL]

Changes the contents of a specified variable or program location.

EVALUATE [/ADDRESS] expression,...

[/DECIMAL | /HEXADECIMAL | /OCTAL]

Evaluates an expression or an address and displays the results in decimal or other specified radix.

EXAMINE variable-reference

[/ASCII:n | /BYTE | /INSTRUCTION | /LONG | /WORD] [/DECIMAL | /HEXADECIMAL | /OCTAL]

[/SYMBOLIC | /NOSYMBOLIC]

Displays the current contents of a variable.

EXIT

Ends the debugging session and returns control to the command interpreter.

GO [%LINE line-number | entry-name | symbolic-reference | nonsymbolic-address]

Starts or continues program execution.

HELP

Displays a description of a debugger command, parameter, or qualifier.

SET BREAK [/AFTER:n] { %LINE line-number | entry-name | symbolic-reference | nonsymbolic-address } [DO (cmd [;cmd...])]

Sets a breakpoint at a specified statement, function, or program address.

SET EXCEPTION BREAK

Requests that the debugger treat external exception conditions as if they were breakpoints; requests a program interrupt when an exception occurs.

SET LANGUAGE language-name

Specifies the source language of a module or routine, for language-specific debugging.

SET LOG [file-spec]

Specifies the name of a log file to which the debugger should write program output when the SET OUTPUT LOG command has been entered.

SET MODE { DECIMAL | HEXADECIMAL | OCTAL | NOSYMBOLIC | SYMBOLIC } , ...

Sets the default mode for entering and displaying program locations that are not declared variables.

SET MODULE { module-name ,... | /ALL }

Adds the symbols from the indicated module(s) to the run-time symbol table.

Table 15–1: (Cont.) Summary of Debug Commands

```
                  ( [ LOG     ]                  )
                  ( [ NOLOG   ]                  )
SET OUTPUT        {                              } , ...
                  ( [ TERMINAL   ]               )
                  ( [ NOTERMINAL ]               )
                  ( [ VERIFY     ]               )
                  ( [ NOVERIFY   ]               )
```

Controls whether the debugger writes output to a log file or to the terminal, and whether it echoes commands executed from command procedures.

```
                  ( 0            )
SET SCOPE         { \            } , ...
                  ( scope-number )
```

Specifies the modules to be searched for a symbol and the order in which they are to be searched.

```
                  ( [ OVER   ]                   )
                  ( [ INTO   ]                   )
SET STEP          {                              }
                  ( [ SYSTEM   ]                 )
                  ( [ NOSYSTEM ]                 )
                  ( [ INSTRUCTION ]              )
                  ( [ LINE        ]              )
```

Specifies how the debugger is to behave when the STEP command is issued.

```
                  ( %LINE line-number  )
                  ( entry-name         )
SET TRACE         { symbolic-reference }
                  ( nonsymbolic-address)
                  ( /BRANCH            )
                  ( /CALL              )
```

Establishes a tracepoint at a specified statement, function, or program location.

Table 15–1: (Cont.) Summary of Debug Commands

SET TYPE [**/OVERRIDE**] { **/ASCII**:length | **/BYTE** | **/INSTRUCTION** | **/LONG** | **/WORD** }

Sets the default data E types for the DEPOSIT and EXAMINE commands for locations that do not have declared data types.

SET WATCH variable-reference

Establishes a watchpoint on a specified static variable.

SHOW BREAK

Displays current breakpoints.

SHOW CALLS [integer]

Displays the current program location and all, or a specified number of, preceding calls.

SHOW LANGUAGE

Displays the current debugging language.

SHOW LOG

Displays the current status of the log file, if any.

SHOW MODE

Displays the current default entry and display modes.

SHOW MODULE

Lists the modules in the image being debugged and shows which modules have names in the run-time symbol table.

SHOW OUTPUT

Displays the current status of the debugger's output files.

SHOW SCOPE

Displays the current default scopes.

SHOW STEP

Displays the current default step conditions.

SHOW TRACE

Displays current tracepoints.

Table 15–1: (Cont.) Summary of Debug Commands

SHOW TYPE [/**OVERRIDE**]

Displays current default data type or override type.

SHOW WATCH

Displays current watchpoints and the number of bytes being watched.

STEP { [/**OVER**] [/**INTO**] } { [/**SYSTEM**] [/**NOSYSTEM**] } { [/**INSTRUCTION** [integer]] [/**LINE** [integer]] }

Executes one or more statements, or steps into or over subroutines.

15.3 Special Characters and Expressions

This section summarizes how the debugger interprets special characters that perform address arithmetic. For example, you can use the multiplication operator (*) in the following manner:

```
DBG>EXAMINE (% LINE 40) *2
```

After this command, the debugger displays the value at the program location whose address is twice that of % LINE 40.

Table 15–2 lists arithmetic operators.

The debugger provides a quick method for referencing relative addresses or locations in DEPOSIT and EXAMINE commands. Table 15–3 lists these relative addressing operators.

15.4 The Run-Time Symbol Table

The debugger maintains a run-time symbol table that lists the symbols you can refer to during a debugging session. The run-time symbol table always contains the names of global symbols in the image. The names of local symbols, that is, names of variables defined within your program, are available in the image file only if you included the /DEBUG qualifier in the CC command.

Table 15-2: Arithmetic Operators

Character	Interpretation
+	Arithmetic addition (binary) operator, or unary plus sign
-	Arithmetic subtraction (binary) operator, or unary minus sign
*	Arithmetic multiplication operator
/	Arithmetic division operator
@	Arithmetic shift operator
< >	Precedence operators; do <enclosed> first
^D	Decimal radix operator
^O	Octal radix operator
^X	Hexadecimal radix operator

Table 15-3: Address Reference Operators

Operator	Meaning
.	The current location (the location most recently referred to by an EXAMINE or DEPOSIT command). Use this symbol with VAX-11 C to refer to a scalar variable or to an element of an array of scalars.
^	The previous location (the location at the next lower address from the current location).
RET	The next location (the location at the next higher address from the current location). Press RET to refer to the next element in an array of scalar variables.

15.4.1 Names Included in the Symbol Table by Default

Before you can refer to a name, you must ensure that the name is in the run-time symbol table. By default, when a debugging session begins, you have access to global symbols and variables declared within the indicated module. For example, a VAX-11 C function may contain the lines:

```
main()
{
     static enum color {red,orange,yellow} c1;
     static char ch;
     static float light_speed;
     static double speed_power;
     static int i;
     static unsigned ui;
     light_speed = 3.0e10;
     speed_power = 3.123456789012345678 9e10;
     c1 = red;
     ch = 'a';
     i = -438394;
     ui = 790374270;
}
```

The debugger identifies the current module as MAIN, and by default you can access the names c1, light_speed, speed_power, i, ui, and ch.[1] When you want to access a variable or location that is not in the symbol table by default, you must specify the module containing the variable or location.

1. The debugger does not recognize case differences between C identifiers. For example, if the function contained the declaration "char C1;", the debugger would not distinguish between the character and enumerated versions of the variable, and it would issue a warning message. In addition, the debugger does not recognize the difference between labels and variables of the same name, even though these do not constitute naming conflicts in VAX-11 C.

15.4.2 Adding Names to the Symbol Table

When a program begins executing, the symbol table contains only the symbols in the first executed function. If you are debugging multiple functions, you must use the SET MODULE command to copy symbols from other modules to the symbol table. For example, a VAX-11 C function can declare an external function as follows:

```
main()
{
      static int i;
      static double f;
      double f2();
      i = 400;
      printf("contents of i: %d\n",i);
      f = f2(i);
      printf("contents of f: %e\n",f);
}

double f2(p)
int p;
{
      static double i;
      static int j;
      i = 3.0e10;
      j = 2;
      return(p*i*j);
}
```

To refer to the variable j in f2, you must first bring f2's symbols into the table with the command:

```
DBG>SET MODULE F2
```

This command makes the names of static variables in f2 accessible. For example, after this SET MODULE command, you can examine j with the command:

```
DBG>EXAMINE J
```

Automatic variables in a function — for example, in f2 — are not accessible until the function actually executes, since it is not until then that variables are allocated storage.

Subsequently, you can use the CANCEL MODULE command to remove symbols you no longer need, and then you can use the SET MODULE command again to insert the symbols you require next. At any time, you can display a list of the available modules with the SHOW MODULE command. For example:

```
DBG>SHOW MODULE
module name      symbols    language    size

MAIN             yes        BASIC        128
F2               yes        BASIC        160
C$MAIN           no         MACRO        104
C$$DOPRINT       no         MACRO        232
C$UNIX           no         MACRO       1988
C$MALLOC         no         MACRO        412
C$CUSERID        no         MACRO        220
C$STRCPY         no         MACRO        168
C$STRLEN         no         MACRO        168
C$STRNCPY        no         MACRO        168
LIB$GET_FOREIGN  no         BLISS        116
LIB$MSGDEF       no         MACRO        104
RMSGBL           no         MACRO        104
total modules: 13.   remaining size: 62992.
```

The display shows all the modules used by the program, 13 in this case, and it shows whether a module's symbols are currently in the symbol table. For this version of the debugger, C modules appear as "BASIC" in the language column. For instance, the symbol table currently contains the symbols in modules MAIN and F2, but not those in the other modules.[1] The symbols in F2 were inserted by the SET MODULE command. Thus, when the program first executes, the "symbols" column for F2 would say "no," meaning that its symbols are not currently accessible.

15.5 Specifying References and Locations

The run-time symbol table lets you refer to names and program locations symbolically. You need concern yourself only with the name, and not the memory location, of the data. This symbolic form of reference applies to scalar variables and to program addresses, such as program line numbers and function names.

You can refer to the following items symbolically:

- Scalar variables (but not function parameters)
- Global symbols
- Program locations
- Symbols you create with the DEFINE debugger command
- Permanent symbols defined by the debugger

1. The module names prefixed by C$, LIB$, RMS, and OTS$$ are run-time modules required for the execution of the VAX-11 C programs.

Symbols can be variable references or values. The debugger interprets them according to the following rules:

1. If a symbol begins with an alphabetic character, the debugger assumes that it is a program variable or a symbolic reference to an address.
2. If a symbol begins with a numeric character (0 through 9), the debugger assumes that it is a numeric constant.
3. If a symbol is enclosed in apostrophes or quotation marks, the debugger assumes that it is a character-string constant.

15.5.1 References to Global Symbols

Global symbols are those symbols defined with the **globaldef** or **globalvalue** storage class keywords. The names of the functions are also considered global symbols. Global symbols can be referenced from all parts of the program.

15.5.2 References to Program Locations

You can refer to program locations by function name, line number, or (nonsymbolic) virtual address. To specify a function by name, give the command followed by the name of the function. For example, the command

```
DBG>SET BREAK LIST_BY_FLOWER
```

sets a breakpoint at the entry to function list_by_flower.

To specify a line number, use the %LINE specifier, as shown here:

```
DBG>SET BREAK %LINE 6
```

This command sets a breakpoint at line 6, which corresponds to the compiler-generated line number shown in the listing.

The debugger does not recognize all line numbers. In particular, it does not recognize those line numbers associated with nonexecutable statements, such as declarations. If you specify such a line number, the debugger responds with a message indicating that no such line exists.

You can also set breakpoints at a line within a function. For example, the commands

```
DBG>SET MODULE LIST_BY_FLOWER
DBG>SET BREAK %LINE LIST_BY_FLOWER\11
```

set a breakpoint at line 11 in list_by_flower.

To specify a virtual address, you issue the command without a prefix. For example:

```
DBG>SET BREAK 700
```

You can determine the virtual address of a line number or a variable by entering an EVALUATE command as follows:

```
DBG>EVALUATE/ADDRESS %LINE 17
800
```

The debugger displays the virtual address of the instructions for the statement on line 17.

15.5.3 Symbolic References to Program Locations

At times you may want to assign a symbolic name to a program location. To do this you must first determine the virtual address of the location with the EVALUATE/ADDRESS command. Then, you must use the DEFINE command to assign the symbolic name. For example:

```
DBG>EVALUATE/ADDRESS %LINE 42
1666
DBG>DEFINE CHK = 1666
```

Subsequent references to line 42 can be made using the defined symbol CHK. For example, the command

```
DBG>SET BREAK CHK
```

sets a breakpoint at line 42. Similarly, the commands

```
DBG>EVALUATE/ADDRESS CARD_COUNTER
6445
DBG>DEFINE CC = 6445
```

define a symbolic name for the variable card_counter.

15.5.4 The Debugger's Permanent Symbols

The debugger has the following permanent symbols; you can use them at any time during the debugging session.

- R0 - R11 General registers 0 through 11
- AP Argument pointer
- FP Frame pointer
- SP Stack pointer
- PC Program counter
- PSL Processor status longword

These names cannot be redefined; for example, you cannot use the name R0 to create a symbol definition with the DEFINE command.[1]

1. The names of permanent symbols may also conflict with identical names used in your program. Furthermore, all names from your program are entered in uppercase in the debugger symbol table, so the name of a variable sp in your program conflicts with the permanent symbol SP.

15.6 Scope

In VAX-11 C, the scope of a name is the function in which the name is declared. If the program you are debugging consists of more than one function, symbolic references may be ambiguous. At times, you may have to tell the debugger how to resolve ambiguous references.

For example, assume that you are debugging two functions; both use a variable i, and both modules are included in the run-time symbol table. Unless you explicitly specify the scope of i, the debugger may be unable to determine which variable i you want.

You can specify the scope in one of three ways:

- By using the debugger's current default scope.
- By explicitly specifying the scope of the variable by prefixing the variable's name with its pathname.
- By setting a new default scope with the SET SCOPE command.

When you begin a debugging session, the debugger automatically defines the first function linked as the default scope (also called the PC scope). However, this default scope is dynamic; that is, as you debug your program, the default scope is always the function that is currently executing. To resolve a symbolic reference, the debugger goes through the following steps:

1. If the specified symbolic name is unique within the run-time symbol table, then the debugger uses that name.
2. If the specified symbol is ambiguous — that is, it is not unique within the symbol table — but one of its occurrences is within the current PC scope, then the debugger recognizes the symbol as it appears in the PC scope.
3. If the specified symbol is not defined in the symbol table, or if it is ambiguous and does not occur within the current PC scope, then the debugger issues an error message indicating that the name is ambiguous.

The program and dialog in Example 15-1 illustrate these rules.

PROGRAM:

```
main()
{
      static int i;
      static double f;'
      double f2();
      i = 400;
      f = f2(i);
}

double f2(p)
int p;
{
      static double i;
      static int j;
      i = 3.0e10;
      j = 2;
      return(p*i*j);
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SHOW MODULE
module name      symbols    language    size

MAIN             yes        BASIC        168
F2               no         BASIC        200
.
.
DBG>EXAMINE i
MAIN\I: 0
DBG>EXAMINE f2\i
%DEBUG-W-NOSYMBOL, symbol 'F2\I' is not in the symbol table
DBG>SET MODULE f2
DBG>EXAMINE f2\i
F2\I:     0.00000000000000000E+00
DBG>SET BREAK %LINE f2\17
DBG>go
routine start at MAIN
break at F2\%LINE 17
DBG>EXAMINE i
F2\I:     30000000000.00000
DBG>EXAMINE main\i
MAIN\I:  400
DBG>EXIT
```

Example 15–1: Scope of Symbolic Names

The first EXAMINE command displays MAIN's version of i, because MAIN is the default scope. The SET MODULE command allows you to examine F2's version of i and to set the breakpoint at line 17 (which is in function f2). The GO command starts the program, which is then interrupted at the breakpoint. Now, the default scope is F2, and the EXAMINE command shows F2\I. At this point, to look at MAIN's version of i, you must use the pathname "main\" in front of the variable name to resolve the reference to i.

When you use a %LINE specifier, the specifier must appear before the pathname. For example:

```
DBG>SET BREAK %LINE SUB1\7
```

This command sets a breakpoint at line 7 in the scope of the module SUB1.

If you want to make frequent references to a location with a long pathname, you can define a symbolic name for it with the DEFINE command. For example:

```
DBG>SET SCOPE INSIDE
DBG>EVALUATE/ADDRESS CARD_COUNTER
9965
DBG>DEFINE CC = 9965
DBG>SET SCOPE MAINP
    .
    .
DBG>EXAMINE CC
```

In this example, the SET SCOPE command changes the scope to the module INSIDE, the EVALUATE/ADDRESS command displays the virtual address of the variable card_counter, and the DEFINE command uses this value to define the symbol named CC. Subsequently, the scope is reset to MAINP. During the debugging session, the value of card_counter can be referred to with the symbolic name CC, regardless of the current scope.

15.6.1 Changing the Scope

If you want to make a number of symbolic references within the same function, you can eliminate the need to specify scope with each symbolic address by using the SET SCOPE command. For example, the following command sets the scope to SUB3:

```
DBG>SET SCOPE SUB3
```

You can also define a scope list to specify the order in which the debugger should search for symbols. For example, the command

```
DBG>SET SCOPE MAR,O,JAN
```

instructs the debugger to search for symbols first in function mar. If it cannot find a specified symbol in mar, then the debugger searches the PC scope, and, if necessary, jan. (The symbol 0 shows that the current scope is the default PC scope.)

The scope defined in a SET SCOPE command becomes the default scope for all symbolic references until you explicitly change or cancel the scope. You can determine the current scope at any time by entering the SHOW SCOPE command. For example:

```
DBG>SHOW SCOPE
scope: MAR ,0, JAN
```

The message shows that the current scope is set first to MAR, then to the PC scope, and finally to JAN.

The SHOW SCOPE command may also respond as follows:

```
DBG>SHOW SCOPE
scope: 0 [ = MULT\MULT]
```

Again, the symbol 0 shows that the current scope is the default PC scope. Within brackets, the debugger displays the module and routine name of the default scope; the scope is module MULT, function mult.

The CANCEL SCOPE command resets the scope to the default PC scope.

When you explicitly SET SCOPE to a function (module) name, the debugger implicitly performs a SET MODULE command. Therefore, symbols for the function specified in your SET SCOPE command are placed in the symbol table. However, if you use the PC scope, you must also use SET MODULE to place symbols for the function in the symbol table.

15.6.2 The Scope of Automatic Variables

If you refer to an automatic variable when the function that defines the variable is not in the current scope, the debugger displays a warning message. For example, this would occur if you tried to refer to an automatic variable declared in a function that has executed a **return** statement, and control has returned to the debugger:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal
                        successful completion'
DBG>EXAMINE  X
%DEBUG-I-PCNOTINSCP, PC is not within the scope of the
                     routine declaring
symbol XLOOK\XLOOK\X:  3
```

This message notifies you that the variable x in the function xlook does not have an address assigned exclusively to it and that its address may have another use in the current section of your program.

15.7 The EXAMINE and DEPOSIT Commands

The EXAMINE and DEPOSIT commands display and change the contents of variables, respectively. When you examine or deposit data into a VAX-11 C variable, you do not need to specify the data type of the variable, unless you want to deposit data of a different type. In the following example, xvalue is of type **float**.

```
DBG>EXAMINE XVALUE
MAIN\XVALUE: 14.50000
DBG>EXAMINE/BYTE XVALUE
MAIN\XVALUE: 68
```

The debugger always uses the declared data type of a scalar variable unless you override it. In this example, the /BYTE qualifier tells the debugger to display only the contents of the first byte of the storage occupied by the variable xvalue.

The SET TYPE/OVERRIDE command tells the debugger to display all variables using a certain type. For example, in response to the following command, the debugger displays only the first byte of any variable's storage:

```
DBG>SET TYPE/OVERRIDE BYTE
```

To restore the normal interpretation of data types, use the CANCEL TYPE/OVERRIDE command.

For this release of VAX-11 C, there are restrictions both on the data types of variables that you can access and on the syntax used to refer to them. Because the language in this version of the debugger is set to BASIC, you cannot examine or modify the following data types in the usual C syntax:

- Arrays
- Structures and unions
- Function parameters

Furthermore, the debugger cannot properly manipulate array elements or structure/union members unless they are integers or characters. The methods for working around the syntactic restrictions are discussed later in this chapter.

15.7.1 Scalar Variables

You can use EXAMINE to display scalar variables of any C data type. If you specify more than one variable and separate them with commas, the contents of each variable are displayed. You can use the DEPOSIT command to change the contents of one variable at a time. Then the variable name and the new value must be separated by an equal sign. The program and debugger dialog in Example 15–2 show how scalar variables of several types are examined and how a new value can be deposited into a variable.

PROGRAM:

```
main()
{
     static float light_speed;
     static double speed_power;
     static unsigned ui;
     static long li;

     light_speed = 3.0e10;
     speed_power = 3.1234567890123456789e10;
     li = -438394;
     ui = 790374270;
}
```

DEBUGGER DIALOG:

```
DBG>SET BREAK %LINE 12
DBG>GO
routine start at MAIN
break at MAIN\%LINE 12
DBG>EXAMINE/DECIMAL li,ui,light_speed,speed_power
MAIN\LI:  -438394
MAIN\UI:  790374270
MAIN\LIGHT_SPEED:     3.0000001E+10
MAIN\SPEED_POWER:       31234567890.12346
DBG>DEPOSIT ui=1
DBG>EXAMINE/DECIMAL ui
MAIN/UI:  1
```

Example 15–2: Examining and Depositing Values in Scalar Variables

15.7.2 Arrays

With the EXAMINE command, you can look at the values in arrays, although the syntax understood and returned by the debugger differs from the usual C syntax for subscripted references. The valid data types for array elements are as follows:

- Integers (all sizes, signed or unsigned)
- Enumerated (**enum**) values[1]

1. In this and all other contexts, the debugger treats enumerated types as integers.

Note that floating-point arrays cannot be examined by the debugger.

The program and dialog in Example 15–3 illustrate the examination of elements in the integer array arr:

PROGRAM:

```
main()
{
      int i;
      static int arr[10];
      for (i=0; i<10; i++)
      {
            arr[i]=i;
      }
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SET BREAK %LINE 8
DBG>GO
routine start at MAIN
break at MAIN\%LINE 8
DBG>EXAMINE arr
MAIN\ARR: 0
DBG>GO
start at MAIN\%LINE 8
break at MAIN\%LINE 8
DBG>GO
start at MAIN\%LINE 8
break at MAIN\%LINE 8
DBG>GO
start at MAIN\%LINE 8
break at MAIN\%LINE 8
DBG>GO
start at MAIN\%LINE 8
break at MAIN\%LINE 8
DBG>EXAMINE arr
MAIN\ARR: 0
DBG>EXAMINE
1028: 1
DBG>EXAMINE
1032: 2
DBG>EXAMINE
1036: 3
DBG>EXIT
```

Example 15–3: Examining Data in an Array

The debugger does not allow you to write BASIC-like subscripted references to the array elements. The dialog in Example 15–3 shows the correct way to look at the elements' values, after several GO commands have executed the loop in the program. The command:

```
EXAMINE ARR
```

shows the contents of the first element (in C syntax, arr[0]). Then, the commands:

```
EXAMINE (RET)
```

show the subsequent locations in the array — the elements arr[1], arr[2], and so on.

In contrast, consider the attempt to examine a floating-point array shown in Example 15–4.

PROGRAM:

```
main()
{
     int i;
     static float arr[10];
     for (i=0; i<10; i++)
     {
          arr[i]=i*3.0e10;
     }
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SET BREAK %LINE 8
DBG>GO
routine start at MAIN
break at MAIN\%LINE 8
DBG>GO
start at MAIN\%LINE 8
break at MAIN\%LINE 8
.
.
DBG>EXAMINE arr
MAIN\ARR: 0
DBG>EXAMINE
1028: -2072620577
DBG>EXIT
```

Example 15–4: Examining Floating-Point Elements of an Array

Here, the array elements are of type **float**, but the debugger displays them as integers. There is no facility in this version of the debugger for displaying elements or members of any aggregate unless they are integral (integers or characters).

15.7.3 Character Strings

To examine or modify parts of a character string, you must override the default mode of the debugger (long integer), by use of the /ASCII qualifier. The program and dialog in Example 15–5 show the EXAMINE and DEPOSIT commands used to examine and change characters in the variable string:

The first EXAMINE command shows a meaningless value for the string, because the debugger uses its default display type, long integer. Notice that you can either specify the /ASCII:n qualifier on each command, where n is the number of characters to be displayed, or you can override the default display type, set it to ASCII:1, and step through the array with EXAMINE commands. With the deposit command, you must use the /ASCII qualifier, or the debugger will interpret the value you try to assign as an integer (hence the warning message).

PROGRAM:

```
#include stdio
main()
{
     char string[20];

     strcpy(string,"VAX-11 C");
}
```

Example 15–5: Examining and Depositing Characters in a Character String

DEBUGGER DIALOG:

```
.
.
DBG>SHOW MODE
modes: symbolic, decimal
type: long integer
type/override: none
DBG>SET BREAK %LINE 65
DBG>GO
routine start at MAIN
break at MAIN\%LINE 65
DBG>EXAMINE STRING
MAIN\STRING: 760758614
DBG>EXAMINE/ASCII:1 string
2147255152: V
DBG>EXAMINE/ASCII:8 string
2147255152: VAX-11 C
DBG>DEPOSIT string = 'PDP'
%DEBUG-W-INVNUMBER, invalid numeric string 'PDP'
DBG>DEPOSIT/ASCII:3 string = 'PDP'
DBG>EXAMINE/ASCII:8 string
2147255152: PDP-11 C
DBG>SET TYPE/OVERRIDE ascii:1
DBG>EXAMINE STRING
2147255152: P
DBG>EXAMINE
2147255153: D
DBG>EXAMINE
2147255154: P
DBG>EXAMINE
2147255155: -
DBG>EXAMINE
2147255156: 1
DBG>EXIT
```

Example 15–5: (Cont.) Examining and Depositing Characters in a Character String

15.7.4 Structures and Unions

You can manipulate structures and unions in a manner similar to that shown for arrays. The program and dialog in Example 15–6 illustrate the examination of structure members:

PROGRAM:

```
main()
{
     static struct
          {
          int im;
          float fm;
          char cm;
          } sv;

     sv.im = -24;
     sv.fm = 3.0e10;
     sv.cm = 'a';
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SET BREAK %LINE 13
DBG>GO
routine start at MAIN
break at MAIN\%LINE 13
DBG>EXAMINE sv.fm
%DEBUG-W-NOSYMBOL, symbol 'SV.FM' is not in the symbol table
DBG>EXAMINE sv
MAIN\SV: -24
DBG>EVALUATE/ADDRESS sv
1024
DBG>EXAMINE
1028: -2072620577
DBG>EXAMINE
1032: 97
DBG>EXAMINE/ASCII:1 .
1032: a
DBG>EXAMINE/ASCII:1 1032
1032: a
```

Example 15–6: Examining Data in Structures

Notice the warning message; the debugger does not accept the C syntax for a structure reference. Instead, the debugger displays the first member in storage, sv.im. Since that member is an integer and the default display mode is integer, the correct value is shown.

The virtual address of sv is shown by the EVALUATE/ADDRESS command, and the following EXAMINE command shows the next item in storage (four bytes away from sv, since the default display mode is a long, or 32-bit, integer). This EXAMINE command shows the contents of the longword at virtual address 1028, which is the correct address of the floating-point member sv.fm, but the value is incorrect. Floating-point members cannot be examined.

The next EXAMINE command shows the next longword, which contains the member sv.cm, a character. You can display it in either decimal or character form, as shown. Note the use of the EXAMINE command to redisplay the contents of the "current" (most recently referenced) location.

Unions are manipulated in a way similar to structures except that all the members of a union occupy the same storage. The dialog in Example 15–7 shows several references to the same location after the SET BREAK and STEP commands have performed the assignment statements individually.

PROGRAM:

```
main()
{
     static union
     {
          int im;
          float fm;
          char cm;
     } uv;

     uv.im = -24;
     uv.fm = 3.0e10;
     uv.cm = 'a';
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SET BREAK %LINE 11
DBG>GO
routine start at MAIN
break at MAIN\%LINE 11
DBG>EXAMINE uv
MAIN\UV: -24
DBG>STEP
start at MAIN\%LINE 11
stepped to MAIN\%LINE 12
DBG>EXAMINE uv
MAIN\UV: -2072620577
DBG>STEP
start at MAIN\%LINE 12
stepped to MAIN\%LINE 13
DBG>EXAMINE uv
MAIN\UV: -2072620703
DBG>EXAMINE/ASCII:1 uv
MAIN\UV: a
DBG>EXIT
```

Example 15-7: Examining Data in Unions

As with structures, the floating-point member, uv.fm, cannot be displayed meaningfully.

15.8 The GO Command

The GO command starts program execution. You use this command when you begin the debugging session and when you want to continue execution after the program has been suspended. For example:

```
$ RUN FLOWERS

                VAX-11 DEBUG Version x.xx


%DEBUG-I-INITIAL, language is 'BASIC', scope and
                  module set to 'FLOWERS'
DBG>GO
    .
    .
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal
                        successful completion'
DBG>
```

The EXITSTATUS message indicates that the program has run to completion.

When you are finished with the debugging session, use the EXIT command to leave the debugger. You must not restart a program from the beginning unless you first exit from the debugger. Otherwise, unpredictable results occur. If your program loops or fails to complete executing, or if you need to interrupt it for any other reason, you can press CTRL/Y to return to the DCL command level. For example:

```
DBG>GO

^Y
$
```

The $ prompt on the terminal indicates that you have returned to the DCL command level. To return to the debugger, type DEBUG or CONTINUE. If you type DEBUG, control returns to the debugger and the debugger prompts you for a command. If you type CONTINUE, the debugging session continues from where it was interrupted.

If you do not want to continue the debugging session, you can enter a STOP command or DCL command to stop the debugging session. You can also reissue the RUN command for the program you are executing, if you want to rerun it from the beginning.

15.9 The STEP Command

You will often want to maintain control of your program so that you can display and/or modify variables after single statements have been executed. The STEP command executes a program one or more lines at a time. For example:

```
DBG>STEP 5
```

causes the debugger to execute the next five statements and suspend the program.

When you are stepping through a program, the debugger displays only the line numbers of the lines as they are executed; it does not display the statements.

The debugger maintains default modes for stepping commands. You can override the default modes with the STEP command qualifiers, or you can change the default with the SET STEP command. For example, the default step for high-level languages is STEP/LINE, indicating a line or statement number increment. In assembly language, the default is STEP/INSTRUCTION. Thus, if you want to look at the machine instructions that are executed for each VAX-11 C statement line, enter the debugger command SET STEP INSTRUCTION, as follows:

```
DBG>SET STEP INSTRUCTION
DBG>STEP
start at MAINP\MAINP\ALPHA %LINE 26
stepped to MAINP\MAINP\ALPHA %LINE 27 : MOVZWL #32,R1
DBG>STEP
start at MAINP\MAINP\ALPHA %LINE 27
stepped to MAINP\MAINP\ALPHA %LINE 27 +3: MOVZWL #32,R3
DBG>STEP
```

For each VAX-11 C statement, there are one or more machine-language instructions. Each STEP command displays the next instruction.

When you subsequently issue a STEP command without qualifiers, the instruction mode remains in effect. You can supersede this default by including the /LINE qualifier in a STEP command. For example:

```
DBG>STEP/LINE 10
```

This command tells the debugger to execute 10 lines, regardless of the current step default.

It is better to use STEP to execute only a few instructions at a time. To execute many instructions and then stop, use a SET BREAK command to set a breakpoint, and then issue a GO command.

15.10 Breakpoints

The BREAK commands let you select locations for suspending the program. Thus, you can let a program run until it reaches a specified statement. Then you can examine and/or modify variables or arrays in the program. The BREAK commands perform the following functions:

- SET BREAK defines a line number, function name, or address at which to suspend execution.
- SHOW BREAK displays all breakpoints currently set in the program.
- CANCEL BREAK removes one or more breakpoints currently set in the program.

For example, the command

```
DBG>SET BREAK %LINE 7
```

sets a breakpoint at the statement corresponding to line number 7 in the compiler listing. The debugger interrupts the program at line 7, *before* the line is executed, as in this example:

```
DBG>SET BREAK %LINE 7
DBG>GO
routine start at MAINP\MAINP
break at MAINP\MAINP %LINE 7
```

After the breakpoint is set, the GO command continues program execution. When statement 7 is reached, the debugger interrupts the program and displays a message indicating that the breakpoint has been reached. At this breakpoint, you can examine or change static variables, begin stepping through the program, and so on.

To set a breakpoint at a function entry point, specify it by name. For example:

```
DBG>SET BREAK PRINT_ROUTINE
```

This command sets a breakpoint at the entry to the function print_routine.

You can use the /AFTER qualifier to control when a breakpoint takes effect. For instance, if you set a breakpoint on a line that is in the range of a loop, you can specify the number of iterations that should be executed before the break occurs, as shown in the following example:

```
DBG>SET BREAK/AFTER:3 %LINE 20
```

In this example, the breakpoint is reported the third time line 20 is encountered and every time it is encountered thereafter.

The SET BREAK command also lets you specify some action to be taken each time a breakpoint is encountered. For example, to set a breakpoint at a location, examine one or more variables, and continue, you could enter a SET BREAK command as follows:

```
DBG>SET BREAK %LINE 29 DO(EXAMINE TOTAL;EXAMINE AREA;GO)
DBG>GO
```

After this command, the debugger sets a breakpoint at line 29. Each time the statement on this line is executed, the debugger interrupts the program, displays the contents of the variables total and area, and executes the GO command to continue execution.

You can cancel a breakpoint with the CANCEL BREAK command. For example:

```
DBG>CANCEL BREAK %LINE 9
```

This command cancels the breakpoint at line 9. To cancel all breakpoints, enter:

```
DBG>CANCEL BREAK/ALL
```

You can display the current breakpoints in effect with the SHOW BREAK command.

15.11 Tracepoints

A tracepoint is similar to a breakpoint in that it suspends program execution and displays the address at the point of suspension. However, in the case of a tracepoint, program execution resumes immediately. Thus, tracepoints let you follow the sequence of program execution to ensure that execution is carried out in the proper order.

Note that if you set a tracepoint at the same location as a current breakpoint, the breakpoint is canceled, and vice versa.

The TRACE commands perform the following functions:

- SET TRACE establishes lines or entry points in the program where execution is to be momentarily suspended.
- SHOW TRACE displays the locations in the program where tracepoints are currently set.
- CANCEL TRACE removes one or more tracepoints currently set in the program.

For example, you can use SET TRACE if you want to keep track of the number of times a given function is called, as follows:

```
DBG>SET TRACE INSIDEOUT
```

Each time a call is made to insideout, the debugger displays a message like the following:

```
routine trace at MAINP\MAINP\INSIDEOUT
```

The message gives the pathname of the symbol.

To set a tracepoint on a given statement, use the %LINE specifier, as in the example below:

```
DBG>SET TRACE %LINE 30
```

While this tracepoint is in effect, the debugger displays a message each time the statement on line 30 is executed.

15.12 Watchpoints

A watchpoint is a location that the debugger monitors. The debugger informs you when your program tries to modify the contents of the location. You can determine, therefore, whether locations are being modified inadvertently during program execution. When you debug a VAX-11 C program, you can set a watchpoint on a variable, and when the watched variable is modified, the debugger suspends program execution, displays the address of the instruction, and prompts for a command.

The following commands control watchpoints:

- SET WATCH defines the location(s) to be monitored.
- SHOW WATCH displays the location(s) currently being monitored.
- CANCEL WATCH disables monitoring of a specified location or of all locations.

You can monitor only static scalar variables. Because automatic variables are allocated storage on the stack, they are protected from access. You cannot set watchpoints, tracepoints, and breakpoints at the same location; the most recently issued command overrides the other.

Run-time errors occur if a watchpoint is in effect while I/O is being performed. Thus, to watch a variable, you must be careful not to set the watchpoint until all previous I/O is completed. You can do this by setting a breakpoint after an I/O statement and then setting a watchpoint. For example, if you want to watch a variable r in a function that contains a **printf** call on line 12, you could set the watchpoint as follows:

```
DBG>SET BREAK %LINE 13 DO (SET WATCH R;GO)
DBG>SET BREAK %LINE 12 DO (CANCEL WATCH R;GO)
```

The SET BREAK commands in the above example ensure that each time **printf** is about to be called, the watchpoint at r is canceled. Following the **printf** call, the watchpoint is reestablished.

When a watched variable is modified, the debugger displays its former contents, if any, and the modified contents. It then prompts you to enter a command. The message is similar to the following:

```
write to MAINP\MAINP\R(1:6) at PC MAINP\MAINP %LINE 13 +25
        old value = +0000000000
        new value = +0054002700
DBG>
```

You must enter GO or STEP to continue the program's execution.

15.13 Entering and Returning from Functions

You can use the following commands to debug a program that consists of more than one function:

- The STEP command lets you specify whether you want to debug a called function or step over it.
- The SHOW CALLS command displays a traceback of the calling sequence.
- The CALL command lets you call a function and pass arguments to it.

15.13.1 Stepping Into and Over Functions

When you are stepping through a program, or when you have set a breakpoint at a function reference, you can decide whether or not to enter the function. To enter the function, type the following command:

```
DBG>STEP/INTO
```

If the names declared in this module are not already in the run-time symbol table, you must also enter a SET MODULE command to include the symbols (including line numbers) to which you want to refer.

If you do not want to debug the function, enter:

```
DBG>STEP/OVER
```

Then, the debugger continues the program's execution at the function's entry point and returns control to you when the function returns.

The STEP command also lets you decide whether you want to step through system functions, such as VAX/VMS system services. If you specify STEP/SYSTEM, then the debugger will step through system functions for you. You cannot, however, set breakpoints or examine data that is being used by system functions.

You can use the SET STEP command to set a default mode for stepping. For example:

```
DBG>SET STEP INTO
```

After this command, the debugger steps into all functions. Note, however, that the debugger steps into the VAX-11 C run-time functions and system services as well as your functions.

15.13.2 Displaying the Calling Sequence

The SHOW CALLS command produces a traceback of calls. This is particularly useful when you have returned to the debugger after a CTRL/Y interrupt. The debugger displays a traceback list that shows you the sequence of calls leading to the current module. If you specify a value, that value determines the number of calls to be displayed. For example,

```
DBG>SHOW CALLS 6
```

causes the six most recent calls to be displayed.

15.13.3 Calling Functions

You can use the CALL command to call a function during the debugging session. You can also specify arguments for the function, although they must be literal constants. The program and debugger dialog in Example 15-8 use the CALL command to call the function f2.

PROGRAM:

```
#define C2 10
#include stdio
main()
{
    static int i;
    i = 400;
    i = f2(i);
}

int f2(p)
int p;
{
    return(p*p);
}
```

DEBUGGER DIALOG:

```
.
.
DBG>SET BREAK %LINE 65
DBG>GO
routine start at MAIN
break at MAIN\%LINE 65
DBG>CALL f2(2)
routine start at F2
value returned is 4
DBG>CALL f2(i)
routine start at F2
value returned is 1048576
DBG>CALL f2(C2)
%DEBUG-W-NOSYMBOL, symbol 'C2' is not in the symbol table
DBG>EXIT
```

Example 15-8: Using the CALL Command

Note that when you specify arguments with the CALL command, you must use only literal constants. Variables are not valid in the argument list of a CALL command, and neither are constants defined by the **#define** control line or **enum** constant names.

The debugger always displays a return value from the function that was invoked. Thus, if the function returns a value, the actual return value will be displayed. However, if the function does not return a value or, as in this example, you specify an invalid argument, the returned value is meaningless.

Appendix A

Portability Considerations

The information in this appendix is not meant to be exhaustive, but rather is meant to serve as a guide to the kinds of portability considerations that C programmers face when porting C programs from different operating systems. This information is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation on the compatibility or functionality of VAX-11 C.

The functions and library routines in Table A-1 are, in general, supported by C compilers. The comments next to each function describe possible differences between the VAX-11 C function and other implementations of the same function.

It is not a goal of VAX-11 C to duplicate all run-time functions that exist on every implementation of the language, but rather to provide a reasonable subset of those functions. Some functions that are available in other implementations have not been implemented on VAX-11 C for any or all of the following reasons:

- The function is not current; recent developments in the language may have made the function obsolete or may have created a replacement for the function.
- The function is incompatible with the VAX/VMS operating system.
- The function would create serious performance restrictions if it were implemented.

Table A-1: Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

abort	VAX/VMS does not generate a core dump.
abs	Equivalent functionality.
access	Equivalent functionality.
acct	Not provided in the VAX-11 C run-time library. The DCL SET command can be used to turn accounting on and off; the VAX/VMS system service, SYS$SNDACC, can be used to send messages to an accounting file.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

acos	Equivalent functionality.
alarm	Equivalent functionality.
alloc	Equivalent functionality.
arc	Not provided.
asctime	Not provided.
assert	Not provided.
asin	Equivalent functionality.
atan	Equivalent functionality.
atan2	Equivalent functionality.
atof	On VAX-11 C, the string may contain any of the white-space characters (space, horizontal or vertical tab, carriage return, form feed, or newline).
atoi	See **atof**.
atol	See **atof**.
brk,sbrk	The VAX-11 C version rounds the break address to the next higher multiple of 512 bytes.
cabs	Equivalent functionality.
calloc	Equivalent functionality.
ceil	Equivalent functionality.
cfree	Equivalent functionality.
chdir	The VAX-11 C version changes the default directory for the user's program only. The user at a terminal will still have the same default directory as before the call. On VAX/VMS, use the DCL SET DEFAULT command. Also, remember the differences between UNIX and VAX/VMS directory syntax.
chmod	VAX/VMS has no equivalent to the "set user id", "set group id" or "save text" file attributes. You can specify group and system read, write, and execute protection individually. **chmod** to 1000 ("save text") is done on VAX/VMS using the INSTALL utility.
chown	Equivalent functionality.
circle	Not provided.
clearerr	Equivalent functionality.
close	Equivalent functionality.
closepl	Not provided.
cont	Not provided.

Table A–1: (Cont.) Relationship of VAX–11 C Run-Time Functions to Other C Run-Time Functions

creat	VAX–11 C adds optional file attributes to allow the creation of files with RMS formats other than stream.
crypt	Not provided.
ctime	Equivalent functionality.
ctype functions	VAX–11 C also has **isgraph** and **isxdigit**. See also the specific ctype (character classification) function in this table.
curses	Not provided.
dbm	Not provided.
dup	Equivalent functionality.
dup2	Equivalent functionality.
ecvt	Equivalent functionality.
endfsent	Not provided.
endgrent	Not provided.
endpwent	Not provided.
erase	Not provided.
exec,execl, execle	The principle of process overlaying is not used in VAX/VMS. On VAX–11 C, you can **exec** C programs only. When specifying the environment array, use the DCL syntax.
exit	If the process was invoked by the DCL command interpreter, then VAX/VMS interprets the return value and prints a DCL message.
exp	Equivalent functionality.
fabs	Equivalent functionality.
fclose	Equivalent functionality.
fcvt	Equivalent functionality.
ferror	Equivalent functionality.
feof	Equivalent functionality.
fdopen	Equivalent functionality.
fflush	Equivalent functionality.
fgetc	Equivalent functionality.
fgets	Equivalent functionality.
fileno	Equivalent functionality.
floor	Equivalent functionality.
fprintf	Equivalent functionality.
fputc	Equivalent functionality.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

fputs	Equivalent functionality.
fopen	File specification must be a valid VAX/VMS file name.
fork	Not provided (see **vfork**).
fread	Equivalent functionality.
free	Equivalent functionality.
freopen	File specification must be a valid VAX/VMS file name.
frexp	Equivalent functionality.
fscanf	VAX-11 C provides the following conversion characters: h, ld, lo, lx, le, and lf.
fseek	When using record files, input from **ftell** is required for VAX-11 C.
ftell	When using record files, VAX-11 C returns the position of the next record.
fwrite	Equivalent functionality.
gamma	Not provided.
gcvt	Equivalent functionality.
getc	Equivalent functionality.
getchar	Equivalent functionality.
getenv	Equivalent functionality.
getgrent	Not provided.
getgrgid	Not provided.
getgrnam	Not provided.
getlogin	Not provided.
getpass	Not provided.
getpw	Not provided.
getpwent	Not provided.
getpwuid	Not provided.
getpwnam	Not provided.
getw	Equivalent functionality.
getfsent	Not provided.
getfsfile	Not provided.
getfsspec	Not provided.
getpid	Equivalent functionality.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

getuid,getgid, geteuid,getegid	VAX-11 C returns the group and member codes from the UIC; VAX/VMS does not distinguish between real and effective user IDs.
gets	Equivalent functionality.
gmtime	Not provided.
hypot	Equivalent functionality.
index	Not provided.
ioctl	Not provided.
isalpha	Equivalent functionality.
isascii	Equivalent functionality.
iscntrl	Equivalent functionality.
isdigit	Equivalent functionality.
islower	Equivalent functionality.
isprint	Equivalent functionality.
ispunct	Equivalent functionality.
isspace	Equivalent functionality.
isupper	Equivalent functionality.
j0,j1,jn	Not provided.
kill	VAX/VMS requires system privileges if the sending and receiving processes have different UICs.
killpg	Not provided.
l3tol	Not provided.
label	Not provided.
ldexp	Equivalent functionality.
link	Not provided.
line	Not provided.
linemod	Not provided.
localtime	On VAX-11 C, daylight savings always equals zero.
log,log10	Equivalent functionality.
longjmp	Equivalent functionality.
lseek	The VAX-11 C version positions on record boundaries for RMS record files.
ltol3	Not provided.
malloc	VAX-11 C aligns the area returned on a longword boundary.

Table A–1: (Cont.) Relationship of VAX–11 C Run-Time Functions to Other C Run-Time Functions

mknod	Not provided.
mktemp	Equivalent functionality.
modf	Equivalent functionality.
monitor	Not provided.
mount,umount	Not provided.
move	Not provided.
mpx	Not provided.
nlist	Not provided. (This information can be obtained from the linker load map.)
nice	On VAX/VMS, the resulting priority cannot be greater than the process base priority.
open	VAX–11 C requires mode = 2 when randomly writing to files.
openpl	Not provided.
pause	On VAX/VMS, processes can also be awakened with the SYS$WAKE system service.
pclose	Not provided.
perror	Equivalent functionality.
pipe	On VAX/VMS, the maximum size of a single write operation is 512 bytes.
point	Not provided.
popen	Not provided.
pow	Equivalent functionality.
printf	Equivalent functionality.
profil	Not provided.
ptrace	Not provided.
putc	Equivalent functionality.
puts	Equivalent functionality.
putw	Equivalent functionality.
qsort	Not provided.
rand	Equivalent functionality.
read	Equivalent functionality.
re_comp	Not provided.
re_exec	Not provided.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

realloc	On VAX-11 C you can reallocate only the last freed area. For example, if you were to make two calls to **free**, only the second area could be reallocated.
reboot	Not provided.
rewind	Equivalent functionality.
rindex	Not provided.
scanf	VAX-11 C provides the following conversion characters: h, ld, lo, lx, le, and lf.
sscanf	VAX-11 C provides the following conversion characters: h, ld, lo, lx, le, and lf.
setbuf	Equivalent functionality.
setgrent	Not provided.
setjmp	Equivalent functionality.
setsfent	Not provided.
setpgrp,getpgrp	Not provided.
setpwent	Not provided.
sighold	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
sigignore	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
signal	Equivalent functionality.
sigpause	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
sigrelse	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
sigset	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
sigsys	Not provided (see VAX-11 C **ssignal,gsignal** routines in Chapter 6).
sin	Equivalent functionality.
sinh	Equivalent functionality.
sleep	Equivalent functionality.
space	Not provided.
sprintf	Equivalent functionality.
sqrt	Equivalent functionality.
srand	Equivalent functionality.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

stat,fstat	Not provided.
stime	Not provided.
strcat	Equivalent functionality.
strcmp	Equivalent functionality.
strcpy	Equivalent functionality.
strlen	Equivalent functionality.
strncat	Equivalent functionality.
strncmp	Equivalent functionality.
strncpy	Equivalent functionality.
swab	Not provided.
sync	Not provided.
syscall	Not provided.
system	Not provided.
tgetent	Not provided.
tgetflag	Not provided.
tgetnum	Not provided.
tgetstr	Not provided.
tgoto	Not provided.
time,ftime	VAX-11 C does not return timezone or daylight fields.
times	VAX/VMS does not distinguish between system and user times. VAX-11 C returns the time in 10-millisecond units.
timezone	Not provided.
tputs	Not provided.
umask	Equivalent functionality.
unlink	Not provided in VAX/VMS. Temporary files can be created using the RMS extensions to **creat**.
ungetc	Equivalent functionality.
utime	Not provided.
va_alist	Not provided.
va_arg	Not provided.
va_dcl	Not provided.
va_end	Not provided.
va_list	Not provided.
va_start	Not provided.

Table A-1: (Cont.) Relationship of VAX-11 C Run-Time Functions to Other C Run-Time Functions

vadvise	Not provided.
valloc	Not provided.
vfork	Equivalent functionality.
vhangup	Not provided.
vlimit	Not provided.
vread	Not provided.
vswapon	Not provided.
vwrite	Not provided.
wait	Equivalent functionality.
wait3	Not provided.
write	Equivalent functionality.

ADDITIONAL NOTES

- The global symbols **end**, **edata**, and **etext** are not implemented in VAX-11 C.
- You should not attempt to substitute your own code for functions that are already supplied by VAX-11 C. For example, the VAX-11 C version of **strcpy** expects a return value. If you were to include a version of **strcpy** which did not return a value, the procedure would not perform correctly. The following code is an example of this:

```
strcpy(p,q) char *p,*q;
{
    while(*p++ = *q++);
}
```

  This use of **strcpy** will not work; there is code inside the VAX-11 C run-time library that expects, and makes use of, a return value.
- Some UNIX-based applications make use of Shell functionality, such as I/O redirection and piping, to achieve program modularity. VAX/VMS DCL does not exactly duplicate this functionality. You may be able to emulate the UNIX Shell I/O redirection capability by setting up the proper values for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR in a DCL command procedure used to run the application(s). For example, the UNIX command

```
% APPLIC1 < FILEA | APPLIC2 > FILEB
```

uses the output file from APPLIC1 as the input file to APPLIC2. The same results can be achieved with the following VAX/VMS DCL commands:

```
$ DEFINE/USER SYS$INPUT FILEA
$ DEFINE/USER SYS$OUTPUT TEMP
$ RUN APPLIC1
$ DEFINE/USER SYS$INPUT TEMP
$ DEFINE/USER SYS$OUTPUT FILEB
$ RUN APPLIC2
```

The same series of commands can be written in a general form (to pass arguments to the DCL command interpreter) in an indirect command file. See Chapter 12 for information on how to create indirect command files.

- There are differences in the way that UNIX and VAX/VMS lay out virtual memory. In UNIX, the address space between 0 and the break address are accessible to the user program. In VAX/VMS, the first page of memory is not accessible.

  If a program tries to reference location 0 on VAX/VMS, a hardware error (ACCVIO) is returned and the program terminates abnormally. VAX/VMS reserves the first page of address space to catch incorrect pointer references, such as a reference to a location pointed to by a null pointer. For this reason, some existing UNIX programs may fail and should be rewritten.

- Some C programmers code all external declarations in **#include** files. Then, specific declarations that require initialization are redeclared in the relevant module. This practice causes the VAX-11 C compiler to issue a warning message about multiply declared/defined variables in the same compilation. One way to avoid this warning is to make the redeclared symbols **extern** variables in the **#include** files.

- **void** is not supported by VAX-11 C in this release.

- The **asm** call is not supported by VAX-11 C.

- Some C programs call the counted string functions **strcmpn** and **strcpyn**. These names are not used by VAX-11 C. Instead, you can define macros that expand the **strcmpn** and **strcpyn** names into the equivalent names **strncmp** and **strncpy**.

- The VAX-11 C compiler does not support the initialization form:

  ```
  int foo 123;
  ```

  Programs using this form of initialization will have to be changed.

- There is a fixed limit to the length of a string that VAX-11 C accepts (1000 bytes). Long strings must be divided, and programs that use string arrays may need to be changed.

- VAX-11 C defines the compile-time constants vax, vms, and vax11c. These constants are useful for programs that must run compatibly on various machines and operating systems. (See Section 7.3.)

- The C language does not guarantee any memory order for the variables in a declaration such as

```
int a,b,c;
```

- The VAX-11 Linker usually places VAX-11 C **extern** variables in program sections (psects) of the same name as the variable. The linker then links the psects alphabetically by name. If you are porting a C program from another operating system to VAX/VMS, you may find that the order of items in the program have been allocated differently in virtual memory. This has caused existing programs with hidden bugs to fail.
- The dollar sign ($) and the underscore (_) are allowed characters in VAX-11 C identifiers.
- VAX-11 C requires the use of VAX/VMS file specifications. See Chapter 12 for a description of the necessary file specification syntax.
- The C language does not define any order for the evaluation of expressions in function parameter lists or in general expressions. The way in which different C compilers evaluate an expression is only important when the expression has "side effects," as in

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

  and

```
f(p++,p++)
```

  Neither VAX-11 C nor any other C compiler can guarantee that such expressions will be portable.
- The size of an integer is 32 bits on VAX-11 C. Programs that were written for other machines and that assume a different size of an **int** will have to be modified.
- The C language defines structure alignment to be dependent on the machine for which the compiler is designed. VAX-11 C aligns structure members on byte boundaries. Other implementations may align structure members differently.
- References to structure members in VAX-11 C must not be ambiguous. (See Section 3.4.2.)
- **case** labels in VAX-11 C must be expressible in 16 bits. Some other implementations may allow **case** labels of different sizes.
- The keyword **register** is ignored by the VAX-11 C compiler; registers are allocated based on how frequently a variable is used. Any scalar variable with the storage class **auto** or **register** may be allocated to a register as long as the variable's address is not taken with the ampersand operator (&) and as long as it is not a member of a structure or union.

- When moving programs from one operating system to another, the operations of the different linkers must also be taken into account. The VAX-11 Linker does not load an object module from an object library unless the module contains a function definition, a **globaldef** (definition), or a **globalvalue** (definition) that is needed to resolve a reference in another component of the program. When you refer to an **extern** variable from a program, the linker does not load the library module if the module contains only a compile-time initialization of the variable. This is a temporary restriction, which can be avoided in either of two ways:

  In the following example, the program PROG.C contains an external declaration of a variable; the module LABDATA.C initializes the variable.

  PROG.C:

```
main()
{
     .
     .
     extern float lab_data[];
     .
     .
}
```

  LABDATA.C:

```
lab_data()
{
     float lab_data = { 1.2, 3.4, 5.6, 7.8 };
}
```

  You could link the object code for the program and the module either by naming the LABDATA object file in the link command, or by explicitly extracting the module from a library (here, it is part of the MYLIB library), as follows:

```
$ LINK PROG,LABDATA,SYS$LIBRARY:CRTLIB/LIB

$ LINK PROG,MYLIB/LIB/INCLUDE=LABDATA,-
$_SYS$LIBRARY:CRTLIB/LIB
```

  You can also bundle the initialization in a module that would be loaded, that is, in a module that contains a function definition, a **globaldef** (definition), or a **globalvalue** (definition).

Appendix B

C Glossary

This appendix defines terms used in this manual.

additive operator
: An operator that performs addition (+) or subtraction (-). It performs the usual arithmetic conversions on its operands.

aggregate
: One of the derived types: array, structure, or union. An array has elements of the same data type. A structure has named members that can be of different data types. A union is essentially a structure that is as long as its longest declared member and that contains the value of only one member at a time.

ampersand (&)
: As a unary operator, computes the address of its operand. As a binary (infix) operator, performs a bitwise AND on two operands, both of which must be of integral type. As an assignment operator (&=), performs a bitwise AND of an expression with the value of the object referred to by the left-hand expression and assigns the result to that object. The double ampersand (&&), a binary operator, performs a logical AND on two operands (see also *logical operator*).

argument
: An expression that appears within the parentheses of a function call. The expression is evaluated and the result is copied into the corresponding parameter of the called function. See also *argument passing* and *parameter*.

argument passing
: The mechanism by which the argument in a function call is associated with a parameter in the called function. In C, all arguments are passed by value; that is, the parameter receives a copy of the argument's value. Therefore, a function called in C cannot modify the value of an argument except via its address. In general, addresses are passed by using the ampersand operator (see *ampersand (&)*) in the argument expression. In addition, use of an array or function name (an array or function identifier with no brackets or parentheses) as an argument always results in the passing of the address of the array or function.

arithmetic operator

A C operator that performs an arithmetic operation. The unary minus (-) operator is at the highest level of precedence. At the next lower level are the binary operators for multiplication (*), division (/), and mod (%). At the next lower level are addition (+) and subtraction (-). There is no unary plus operator, and there is no exponentiation operator. All the binary operators perform the usual arithmetic conversions on their operands.

arithmetic type

One of the integral data types, enumerated types, **float**, or **double**.

array

An aggregate data type consisting of subscripted elements of the same type. Elements of an array can have one of the fundamental types or can be structures, unions, or other arrays (to form multi-dimensional arrays).

assignment expression

An expression of the form:

```
E1 asgnop E2
```

where E1 must be an lvalue, asgnop is an assignment operator, and E2 is an expression. The type of an assignment expression is that of its left operand. The value of an assignment expression is that of the left operand after the assignment has taken place. If the operator is of the form "op=", then the operation E1 op (E2) is performed, and the result is assigned to the object referred to by E1; E1 is evaluated only once.

assignment operator

The combination of an arithmetic or bitwise operator with the assignment symbol (=); also, the assignment symbol by itself. These operators are used in assignment expressions.

asterisk (*)

As a unary operator, treats its operand as an address and results in the contents of that address. As a binary operator, multiplies two operands, performing the usual arithmetic conversions. As an assignment operator (*=), multiplies an expression by the value of the object referred to by the left operand, and assigns the product to the object.

binary operator

An operator that is placed between two operands. The binary operators include arithmetic operators, shift operators, relational operators, equality operators, bitwise operators (AND, OR, and XOR), logical connectives, and the comma operator, in that order of precedence. All binary operators group from left to right. (Note: C has no operator for exponentiation.)

bitwise operator

An operator that performs a bitwise logical operation on two operands, which must be integral. The usual arithmetic conversions are performed. Both operands are evaluated. All bitwise operators are associative, and expressions using them may be rearranged. The set comprises, in order of precedence, the single ampersand ([&] bitwise AND), the circumflex ([^] bitwise exclusive OR), and the single bar ([|] bitwise inclusive OR).

block

See *compound statement.*

block activation

The run-time action of activating a block or function, in which local **auto** and **register** variables are allocated storage and, if they are declared with initializers, given initial values. (**static**, **extern**, **globaldef**, and **globalvalue** variables are allocated and initialized at compile time.) The block activation precedes the execution of any executable statements in the function or block. Although it is not literally true, you can think of a block activation as the "execution of the declarations" in the block. Functions are activated when they are called. Internal blocks (compound statements) are activated when the program control flows into them. Internal blocks are not activated if they are entered by a **goto** statement, unless the **goto** target is the label of the block rather than the label of some statement within the block. If a block is entered by a **goto** statement, references to **auto** and **register** variables declared in the block are still valid references, but the variables may not be properly initialized. Blocks which make up the body of a **switch** statement are not activated; **auto** or **register** variables declared in the block are not initialized.

cast

An expression preceded by a cast operator of the form "(typename)". The cast operator forces the conversion of the evaluated expression to the given type. The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction. The cast operator has the same precedence as the other unary operators.

character

(1) A member of the ASCII character set.

(2) An object of the C data type **char** — that is, a byte. (An object of type **char** always represents a single character, not a string.)

(3) A constant of type **char**, consisting of up to four ASCII characters enclosed in apostrophes (′, not ").

See also *string.*

comma operator

A C operator used to separate two expressions:

```
E1 , E2
```

The expressions E1 and E2 are evaluated left to right, and the value of E1 is discarded. The type and value of the comma expression are those of E2.

comment

A sequence of characters introduced by the pair /* and terminated by */. Comments are ignored during compilation. They may not be nested.

compound statement

A compound statement consisting of valid C statements enclosed in braces ({}). Compound statements can also include declarations. The scope of these variables is local to the block.

conditional operator

The C operator (?:), which is used in conditional expressions of the form:

```
E1 ? E2 : E3
```

where E1, E2, and E3 are expressions. E1 is evaluated, and if it is nonzero, the result is the value of E2; otherwise, the result is the value of E3. Only one of E2 and E3 is evaluated.

constant

A primary expression whose value does not change. A constant may be literal or symbolic.

constant expression

An expression involving only constants. Constant expressions are evaluated at compile time and may therefore be used wherever a constant is valid.

conversion

The changing of a value from one data type to another. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; that type is also the type of the assignment expression. Conversions are also performed when arguments are passed to functions: **char** and **short** become **int**; **unsigned char** and **unsigned short** become **unsigned int**; **float** becomes **double**. Conversions can also be forced by means of a cast. Conversions are performed on operands in arithmetic expressions by the usual arithmetic conversions.

data definition

The syntax that both declares the data type of an object and reserves its storage. For variables that are internal to a function, the data definition is the same as the declaration. For external variables, the data definition is external to any function (an external data definition).

declaration
A statement that gives the characteristics (such as data type) of one or more variables.

enumerated type
A type defined (with the **enum** keyword) to have an ordered set of integer values. The integer values are associated with constant identifiers named in the declaration. Although **enum** variables are stored internally as integers, they should be used in programs as if they had a distinct data type.

equality operator
One of the operators == (equal to) or != (not equal to). They are analogous to the relational operators, but at the next lower level of precedence.

exponentiation operator
The C language does not provide an exponentiation operator.

expression
A series of tokens that the compiler can use to produce a value. Expressions have one or more operands and, usually, one or more operators. (An identifier with no operator is an expression that yields a value directly.) Operands are either identifiers (such as variable names) or other expressions, which are sometimes called subexpressions. See also *operator*.

external variable
A variable that is defined externally to any function. External variables provide a means other than argument passing for exchanging data between the functions that comprise a C program.

floating type
One of the data types **float** or **double**, representing a single- or double-precision floating-point number.

function
The primary unit from which C programs are constructed. A function definition begins with a name and argument list, which are followed by the declarations of the arguments (if any) and the body of the function enclosed in braces ({ }). The function body consists of the declarations of any local variables and the set of statements that perform its action. Functions need not return a value to the caller. All C functions are external; that is, a function may not contain another function. See also *function call.*

function call
A primary expression followed by parentheses. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. Any previously undeclared identifier followed immediately by parentheses is contextually declared as a function returning **int**. Any function may call itself recursively.

fundamental type
The set of arithmetic data types plus pointers. In general, the fundamental types in C comprise those data types that can be represented naturally on a particular machine; usually, this means integers and floating-point numbers of various machine-dependent sizes, and machine addresses.

identifier
A sequence of letters and digits, the first 31 of which must be unique. The underscore (_) and dollar sign ($) are letters in this context. The first character of an identifier must be a letter. Upper- and lowercase letters specify different identifiers in VAX-11 C. Note, however, that all external names are converted to uppercase to be consistent with VAX/VMS.

initializer
The part of a declaration that gives the initial value(s) for the preceding declarator. An initializer consists of an equal sign (=) followed by either a single expression or a comma-separated list of one or more expressions in braces.

integral type
One of the data types **char** or **int** (all sizes, signed or unsigned).

keyword
A word (series of characters) that is reserved by the C language and cannot be used as an identifier. Keywords identify statements, storage classes, data types, and the like. Function names are not C keywords; they may be redefined by the user.

literal
A constant whose value is written explicitly in the program. Literals have type **int** or **double**, depending on their forms. Character constants have type **int**. Floating constants have type **double**. Character-string constants have type "array of" **char**.

logical expression
An expression made up of two or more operands separated by logical connectives. Each operand must be of a fundamental type or must be a pointer or other address expression. Operands do not have to be of the same type. Logical expressions always return 1 or 0 (type **int**) to indicate a true or false value, respectively. Logical expressions are always evaluated from left to right, and the evaluation stops as soon as the result is known.

logical operator
One of the binary operators && (logical AND) and || (logical OR).

lvalue

The abstract value that denotes the location of an object whose contents can be assigned or modified. In this manual, the term is used to describe a category in C grammar. An lvalue is required on the left-hand side of an assignment operator (hence its name) and as the operand of certain other operators, such as the increment (++) and decrement (--) operators. A variable name is an example of an lvalue, since its address can be taken (with &), and values can be assigned to it. A constant is an example of an expression that is not an lvalue.

macro

A text substitution that is defined with the **#define** preprocessor control line and includes a list of "parameters." The parameters in the **#define** control line are replaced at compile time with the corresponding arguments from a macro reference encountered in the source text.

multiplicative operator

An operator that performs multiplication (*), division (/), or modulo arithmetic (%). It performs the usual arithmetic conversions on its operands. The mod operator (%) yields the remainder of the division of the first operand by the second.

object

One of the basic elements that the language can manipulate — that is, the elements to which operators can be applied. In C, objects include data (such as integers, real numbers, or characters), data structures (arrays, structures, unions), and functions.

operator

A token that performs an operation on one or more operands. In order of precedence (high to low), operators are classified as the primary-expression operators, unary operators, binary operators, the conditional operator, assignment operators, and the comma operator.

parameter

A variable declared in an external function definition, between the function name and the body of the function. The parameter receives a copy of the value of an associated argument when the function is called. The items in parentheses in a macro definition are also called parameters, although the semantics are different from C function calls.

pointer

A variable that contains the address of another variable or function. A pointer is declared with the unary asterisk operator.

preprocessor control lines

Lines of text in a C source file that change the order or manner of subsequent compilation. The control lines are **#define** (for macro substitution and other token replacements), **#undef** (to cancel a previous **#define**), **#include** (for inclusion of external source text), **#line** (to specify a line number to the compiler), **#module** (to specify a module name to the linker), and **#if**, **#ifdef**, **#ifndef**, **#else**, and **#endif** (to conditionalize the compilation of the program). In VAX-11 C, these control lines are processed by an early phase of the compiler, not by a separate program.

primary expression

An expression that contains only a primary-expression operator, or no operator. Primary expressions include previously declared identifiers, constants, strings, function calls, subscripted expressions, and references to structure or union members.

primary-expression operator

A C operator that qualifies a primary expression. The set of such operators consists of paired brackets (to enclose a single subscript), paired parentheses (to enclose an argument list or to change the associativity of operators), a period (to qualify a structure or union name with the name of a member), and an arrow (to qualify a structure or union member with a pointer or other address-valued expression).

relational operator

One of the operators <, >, <=, or >=. The result (type **int**) is 1 or 0, indicating a true or false relation, respectively. The usual arithmetic conversions are performed on the two operands. Relational operators group from left to right.

scalar

A single object (as opposed to *aggregate*). See also *object*.

scope

The portion of a program in which a particular name has meaning. The scope of names declared in external definitions extends from the point of the definition's occurrence to the end of the compilation unit in which it appears. The scope of the names of function parameters is the function itself. The scope of names declared in any block (that is, after the brace beginning any compound statement) is restricted to that block. Names declared in a block supersede any other declaration of the name, including external definitions, for the extent of that block. **struct**, **union**, **typedef**, and **enum** tags are identifiers that are subject to the same scope rules as any identifiers. Member names in structure or union references are not subject to the same scope rules (see *uniqueness*). The scope of a label is the entire function containing the label.

shift operator
One of the binary operators << or >>. Both operands must have integral types. The value of E1<<E2 is E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1>>E2 is E1 right-shifted by E2 bits.

statement
The language elements that perform the action of a function. Statements include expression statements (an expression followed by a semicolon), null statements (the semicolon by itself), compound statements (blocks), and an assortment of statements identified by keywords (such as **return**, **switch**, **do**).

storage class
The attribute that, with its type, specifies C's interpretation of an identifier. The storage class determines the location and lifetime of an identifier's storage. Examples are **static**, **external**, and **auto**.

string
(1) An array of type **char**.

(2) A constant consisting of a series of ASCII characters enclosed in quotation marks. Such a constant is declared implicitly as an array of **char**, initialized with the given characters, and terminated by a NUL character (ASCII 0, C escape sequence \0).

structure
An aggregate type consisting of a sequence of named members. Each member may have any type. A structure member may also consist of a specified number of bits, called a field.

symbolic constant
An identifier assigned a constant value by a **#define** control line. A symbolic constant may be used wherever a literal is valid.

tokens
The fundamental elements making up the text of a C program. Tokens are identifiers, keywords, constants, strings, operators, and other separators. White space (such as spaces, tabs, newlines, and comments) is ignored except where it is necessary to separate tokens.

type
The attribute that, with its storage class, specifies C's interpretation of an identifier. The type determines the meaning of the values found in the identifier's storage. Types include the integral and floating types, pointers, enumerated types, and the derived types array, function, structure, and union.

type name
In essence, the declaration of an object of a given type that omits the name of the object. A type name is used as the operand of the cast and **sizeof** operators.

unary operator
An operator that takes a single operand. In C, some unary operators can be either prefix or postfix. The set includes the asterisk (indirection), ampersand (address of), minus (arithmetic unary minus), exclamation (logical negation), tilde ([~] one's complement), double plus (increment), double minus (decrement), cast (force type conversion), and **sizeof** (yields size, in bytes, of its operand).

union
An aggregate type. It can be considered a structure all of whose members begin at offset 0 from the base and whose size is sufficient to contain any of its members.

uniqueness
A property of the names used for certain structure and union members. A name is unique if either of these conditions is true:

- The name is used only once.
- It is used in two or more different structures (or unions), but each use denotes a member at the same offset from the base and of the same data type.

The significance of uniqueness is that a unique member name can be used to refer to a structure in which the member name was not declared (although a warning message is issued).

usual arithmetic conversions
The set of rules that govern the conversion of operands in arithmetic expressions. The rules are applied in the following order:

1. Any operands of type **char** or **short** are converted to **int**, and any of type **float** are converted to **double**.
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**, the other is converted to **unsigned** and that is the type of the result.
4. Otherwise, both operands must be **int**, and that is the type of the result.

variable
An identifier used as the name of an object.

Appendix C

VAX–11 C Compiler Messages

This appendix lists the VAX–11 C compiler diagnostic messages alphabetically. For each message, the appendix gives the mnemonic, the message text, an explanation of the message, and suggested actions to be taken to avoid the message. Chapter 14 gives information on the format of compiler messages.

You can also obtain the compiler diagnostic messages online. Type:

```
$ HELP ERROR CC mnemonic (RET)
```

To receive a list of all the mnemonics, type:

```
$ HELP ERROR CC (RET)
```

Some messages substitute information from the program in the message text. In this appendix, the portion of the text to be substituted is shown as "*****" or ****. If quotes appear around the asterisks, quotes appear in the substituted message.

ANACHRONISM, The "****" operator is an obsolete form, and may not be portable.

Warning. You have used an old-style operator such as =+ or =*. The message is issued only if the /STANDARD=PORTABLE qualifier is specified in the CC command line.

User Action. For the program to be portable, you should reverse the order of the parts of the operator. For example, =+ should be += and =* should be *=. The old-style operators are supported by VAX–11 C, but they may not be supported by other C compilers.

ARGLIST, The VAX architecture does not allow argument lists to be more than 255 longwords (ints) in length. The arguments beyond this limit have been ignored.

Warning. You have called a function with too long an argument list.

User Action. To avoid this message and unpredictable results, rewrite the function definition and function call with a shorter argument list.

ARGNOTPORT, Passing a structure by value is not portable.

Warning. This message occurs when a structure is passed by value in a function call, or when a function parameter is declared as a structure, and when the /STANDARD=PORTABLE option is used in the CC command line.

User Action. If the program must be portable, pass the structure, by reference, as a pointer to the structure.

ARGOVERFLOW, Length of macro argument list exceeds buffer capacity; overflowing argument(s) considered to be null.

Warning. The total length of the arguments in a macro reference exceeds the compiler's capacity to store the arguments prior to substitution.

User Action. To avoid this message and unpredictable results, shorten one or more arguments.

ASNDATTYPE, Target of assignment operator is noncomputational data type.

Error. You have specified, as the left-hand operand of an assignment operator, an expression that is not valid for assignment. For example, you have tried to assign something to an array or function.

User Action. Correct the statement. The assignment target must be a scalar variable (including a scalar array element or structure member), structure, union, or dereferenced pointer.

ASNREADONLY, Target of assignment operator is read-only.

Error. You have attempted to assign a value to a read-only object. Such objects include literal constants, enumerated constants, and variables declared with the storage class **readonly**.

User Action. If you specified a constant, replace it with a valid assignment target. If you specified a **readonly** variable, remove the **readonly** keyword from its definition.

BADAUTOINIT, The automatic initialization for "****" is not valid. It has been ignored.

Warning. Automatic character arrays cannot be initialized with string constants. This is a temporary restriction.

User Action. Initialize the array with an explicit list of characters, or copy the string to the array using the strcpy function.

BADCODE, Invalid code generation sequence.

Fatal. An internal compiler error occurred.

User Action. Gather as much information as you can about the conditions in effect when the error occurred, and submit an SPR.

BADCONDEXPR, The nonpointer operand of a conditional expression must be the integer constant 0.

Error. Conditional expression operands involving pointers must consist of either one or two pointers and the integer constant 0. For example:

```
e1?(int*) p: 1.2;
```

The **float** constant 1.2 is not allowed.

User Action. To avoid this message, replace the invalid operand with an expression of the correct type (pointer or 0).

BADIFEVAL, **** while evaluating #if expression; "true" expression assumed.

Warning. The substitute text is "Stack overflow", or "Divide by zero". The expression was taken to be true.

User Action. Reevaluate the line and make the appropriate corrections.

BADMODULE, Redundant #module control line; line ignored.

Warning. You specified more than one **#module** control line in a single compilation; the excess line or lines were ignored.

User Action. To avoid this message correct the lines.

BADPARDCL, "****" is not a named formal parameter in the defined function. It has been declared as auto.

Warning. You declared the specified identifier as a function parameter, and the identifier does not appear in the parameter list. For example:

```
f(a) int a,b; {...}
```

The identifier b was declared as an **auto** variable in the defined function.

User Action. The identifier's declaration is nonportable and is possibly a programming error. To avoid this message and unpredictable results, correct the declaration or function definition.

BADPSECT, The program section (psect) specified by this statement has conflicting READONLY attributes with another definition of the same program section.

Warning. You have specified two or more references to the same program section, and they do not agree with the program section's attributes. For example, this message can appear when two **globaldef** definitions appear for the same name, but only one specifies the storage class **readonly**.

User Action. To avoid this message and unpredictable results, make all references to a program section consistent.

BADSUBVAL, Array subscripts must be specified for subscripts other than the first.

Warning. You omitted too many subscripts from an array declaration. A subscript (the number of elements) can be omitted if there is only one dimension. If there is more than one dimension, only the first (leftmost) pair of brackets in the declaration can be empty. This applies equally to declarations of array parameters and **extern** declarations of arrays.

User Action. To avoid this message and unpredictable results, correct the declaration, specifying a size for every dimension after the first.

BADUNARY, The operand of a **** operator was noncomputational.

Error. You specified an expression of an illegal type with a unary operator, such as ++function-call or --union.

User Action. Replace the operand with an expression valid for the illustrated operator.

BADVINIT, "****" is a value. It may be initialized only with a constant expression.

Warning. You attempted to initialize a **globalvalue** with a nonconstant initializer.

User Action. To avoid this message and unpredictable results, correct the initializer.

BINNONCOMPUT, The **** operand of a "****" operator is noncomputational.

Error. Either the left or right operand of the illustrated operator is an illegal data type, as in function-name+2.

User Action. Specify an expression that is valid for use with the operator.

BUGCHECK, Compiler bug check during ****. Submit an SPR with a problem description.

Fatal. An internal error occurred during the specified phase of compilation.

User Action. Gather as much information as possible about the conditions under which the error occurred, including the phase of compilation, and submit an SPR.

CASECONSTANT, Case label value is not a constant expression.

Error. You specified a value in a **case** label that was not a constant.

User Action. Replace the **case** value with a valid constant expression.

CASERANGE, Case label value **** is not a 16-bit integer.

Error. You specified a value in a **case** label that is not expressible in 16 bits.

User Action. Correct the **case** label so that it is expressible in 16 bits.

CMPLXINIT, "*****" is too complex to initialize.

Warning. The depth of the indicated aggregate variable exceeds the limit of 32 levels.

User Action. To avoid this message and unpredictable results, simplify or correct the initializer list or declaration. Otherwise, initialize the variable with explicit assignments.

COMPILERR, Previous errors prevent continued compilation. Please correct reported errors and recompile.

Fatal. The compiler has detected too many errors to continue.

User Action. Correct the errors reported in the compiler messages previous to this one.

CONDEXPRPTR, The second and third operands of a conditional expression, if pointers, must be pointers to objects of the same type (size).

Error. The two operands must be pointers to the same data type because the result of such a conditional expression is an object of the common type.

User Action. Correct the operands accordingly.

CONFLICTDECL, This declaration of "****" conflicts with a previous declaration of the same name.

Warning. A name has been redeclared, and the data types and/or organizations are different. Any reference to the name resolves to the most recent declaration, according to the scope rules.

User Action. The purpose of this message is to call a possible programming error to your attention.

DEFTOOLONG, Text in #define control line is too long; line ignored.

Warning. The total number of symbols in the **#define** line exceeds the implementation's limit.

User Action. To avoid this message and unpredictable results, shorten or otherwise simplify the line.

DIVIDEZERO, Constant expression includes divide by zero; the result has been replaced with 0.

Warning. A division by zero was encountered in a constant expression. The expression was replaced by 0.

User Action. Check your constant expressions and correct the one that is causing the error.

DUPCASE, Case label value **** is a duplicate.

Error. You have specified more than one **case** for the indicated value in a **switch** statement. The cases must be unique.

User Action. Change the **case** labels and/or combine the cases, as appropriate.

DUPLDEF, Duplicate definition of "****".

Warning. The named definition appears more than once in the program. The two definitions are essentially the same. Both definitions specify the same data types and organizations, but there may be differences in the values, initializers, or array bounds. If the name is a function, there may be a difference in the number or types of parameters, or in the contents of the function body.

User Action. The purpose of this message is to call a possible programming error to your attention.

DUPDEFAULT, Duplicate default label in switch statement.

Error. You specified more than one **default** case in the same **switch** statement.

User Action. Combine the cases or make other changes necessary to eliminate the duplicate(s).

DUPLICATE, Duplicate label "****".

Error. You have specified duplicates of the indicated label in the same function. Label identifiers must be unique within a function definition.

User Action. Rewrite the labels (and **goto** statements that refer to them) to eliminate the duplicates.

DUPLPARM, Duplicate macro parameter "****" ignored.

Warning. The indicated macro parameter occurred more than once in the **#define** line's parameter list. All instances of it after the first were ignored.

User Action. To avoid this message and unpredictable results, correct the line.

ENUMOP, Mismatched enum type in "****" operation.

Warning. The indicated operation combines an **enum** variable or value with a value of a nonmatching type.

User Action. To avoid this message and unpredictable results, use a cast operation to cast either the **enum** value or the other value to a matching type.

ERRORSUM, Completed with severe errors.

Fatal. The compilation is complete, but there were too many errors to produce an object file.

User Action. Correct the errors reported in the previous compiler messages.

EXTRAARGS, Too many arguments specified for a function reference. Only the first 253 will be passed.

Warning. You have called a function with more than 253 arguments.

User Action. To avoid this message and unpredictable results, shorten the argument list.

EXTRACOMMA, Extraneous comma in macro parameter list ignored.

Warning. The **#define** macro definition on this line has extra commas that were ignored.

User Action. To avoid this message and unpredictable results, correct the line.

EXTRATEXT, Extraneous text in preprocessor control line ignored.

Warning. Extraneous text appears in the control line, as in

```
#endif ABC
```

User Action. To avoid this message, correct the line.

FATALSYNTAX, Fatal syntax error.

Fatal. The compiler cannot continue due to syntax errors.

User Action. Correct the error in the indicated line and/or errors reported in previous compiler messages.

FIELDBADSIZE, "****" is an invalid field declaration. Fields may be up to 32 bits in length. Size of 32 bits assumed.

Warning. The indicated field declaration is invalid because it specifies too large a size.

User Action. To avoid this message and unpredictable results, correct the declaration to specify either a single, smaller field or several contiguous fields. Note, however, that field alignments are nonportable.

FIELDBADTYPE, "****" is an invalid field declaration. Fields must be declared as integers (signed or unsigned) or enum. The data type int has been assumed.

Warning. You have declared a field with an invalid data type. Fields must be declared (and manipulated) as integers or enumerated types.

User Action. To avoid this message and unpredictable results, correct the declaration to specify a valid data type.

FILEUNOPEN, Unable to open the **** file.

Fatal. The compiler cannot continue because of the failure to open the indicated file.

User Action. Be sure that the file exists if it is an input file, or change the file specification in the program to that of an existing file.

FNDUPLPARM, Duplicate function formal parameter "****" ignored.

Warning. The stated function parameter occurred more than once in the function's formal parameter list, as in

```
funct(a,b,c,a) { }
```

All occurrences of the parameter after the first were ignored.

User Action. To avoid this message and unpredictable results, correct the line.

FNPARMREDECL, Function formal parameter "****" has been redeclared.

Warning. Your source program contains a redeclaration of one of the function's formal parameters, as in:

```
f(a) { int a; }
```

User Action. Verify that this is what you want to do. If it is not, correct the declaration(s).

FUNCBADECL, "****" is not properly declared. The function attribute will be ignored.

Warning. You have used the "function attribute" in an illegal manner in the definition of the indicated object; for example, you have declared an array of functions instead of an array of pointers to functions. The function attribute was dropped.

User Action. Check the definition carefully to be sure that it is logical and is what you intended. In general, you should correct definitions that raise this warning.

IFNOMACSUB, Macro substitution cannot be performed during the scan of a macro reference; "****" not substituted; "true" expression assumed.

Warning. You have written a complex macro reference with an **#if** control line and another macro reference, as in:

```
macref(arg1,
#if SUBST
        ,
        ,
        ,arg2)
```

where SUBST has a substitution defined in a previous **#define** line. The substitution (here, for SUBST) was not performed, and the truth value of the control line was assumed to be true.

User Action. To avoid this message and unpredictable results, replace the reference to the macro in the **#if** expression with its actual value, or restructure the **#if** construct so that it is not within the complex macro reference.

IFSYNTAX, Syntax error in #if expression; true expression assumed.

Warning. The **#if** expression on the indicated line cannot be evaluated because of syntax errors; it is assumed to be true.

User Action. To avoid this message and unpredictable results, correct the line.

INCDECTAR, The operand of a **** operator is not an lvalue.

Error. You have specified an invalid operand with the -- or ++ operator. The operand must be an lvalue, such as a variable reference or a dereferenced pointer.

User Action. Replace the operand with an lvalue.

INCMODNOTPORT, #include of a library module is not portable.

Warning. The specification of a library module name in an **#include** preprocessor control line is a VAX-11 C extension and is not portable. This message is issued only if the /STANDARD=PORTABLE qualifier is specified on the CC command line.

User Action. No action is necessary if you do not require compatibility with other C compilers.

INCNESTLVL, Include files may only be nested 4 levels.

Fatal. You have specified a tree of **#include** files or modules that are too deeply nested. The implementation limit is four.

User Action. If you need all the text you specified in the **#include** lines, explicitly include some of it in the source file.

INCOMDT, **** is an invalid data type in this declaration. All but the first data type ignored.

Warning. You specified the indicated data type keyword in a declaration or definition that already had one, as in **float int**. The resulting type in this example is **float**.

User Action. Check carefully to see which type the object should have, and remove all the extraneous keywords.

INCOMSC, **** is an invalid storage class in this declaration. Only the first storage class is used.

Warning. The indicated storage class keyword appears in a declaration that already has one, as in **auto register**. Only the first one (here, **auto**) is used.

User Action. To avoid this message and unpredictable results, correct the declaration.

INCPTRSUB, Inconsistent pointer subtraction; two pointers may be subtracted only if they point to equivalent-sized objects.

Error. You subtracted two pointers that do not point to the same data type or to objects of equal size. The subtraction is an invalid operation.

User Action. Change or cast the operands to point to equal-sized objects.

INITBIT, "****" is a field. Static fields may be initialized only with constants.

Warning. You have initialized the indicated structure field with a variable.

User Action. The field may not be properly initialized. To avoid this message and unpredictable results, specify a constant initializer.

INVADDR, Invalid "address of" operand.

Error. You have used the "address of" (&) operator with an invalid operand. The operand must be an lvalue, such as the name of a variable, and it must not be a reference to a bit field.

User Action. Correct the operand.

INVAGGASN, Invalid aggregate assignment; union = union and structure = structure are valid if the source and the target are of equal size.

Error. You have attempted to assign an array to another array or to assign structures or unions of different sizes.

User Action. Correct the assignment.

INVALIDIF, "****" is not a valid constant or operator in an #if expression; "true" expression assumed.

Warning. You have used an invalid construction in an **#if** expression, which is assumed to be true.

User Action. To avoid this message and unpredictable results, correct the line.

INVALINIT, The initialization of "****" is not valid.

Warning. The indicated object cannot be initialized as specified. Some objects may not be initialized at all, such as functions, unions, and **extern** or **globalref** objects. In other cases, the initializer may not be appropriate, for example, a static pointer cannot be initialized with the address of an automatic variable. This and any subsequent initializers for the same object have been ignored.

User Action. To avoid this message and unpredictable results, eliminate or correct the initializer, or correct the type or storage class of the target object, or initialize the object with an explicit assignment.

INVARRAY, "****" is an improperly declared array.

Warning. You have improperly declared an array, such as an array of functions.

User Action. The declared object is probably not what you wanted. To avoid this message and unpredictable results, correct the declaration.

INVBITARR, Fields cannot be subscripted.

Warning. You have specified subscripts in the declaration of a field. There cannot be arrays of fields.

User Action. To avoid this message and unpredictable results, correct the declaration.

INVBREAK, break statement used in an invalid context. break is valid only in for, while, do, and switch statements.

Error. You have used **break** outside the body of any of the indicated statements.

User Action. Remove the offending **break**.

INVCMDVAL, "****" is an invalid command qualifier value.

Warning. The indicated CC command qualifier value was acceptable to the VAX/VMS command interpreter (CLI), but it is meaningless to VAX-11 C; for example, LIST_OPTS is an invalid value for /SHOW, although it is accepted by the CLI.

User Action. Correct the qualifier value.

INVCONST, "****" is an invalid numeric constant.

Warning. The indicated constant has illegal characters or is otherwise invalid.

User Action. To avoid this message and unpredictable results, correct the constant.

INVCONTINUE, continue statement used in an invalid context. continue is valid only in for, while, and do statements.

Error. You have used the **continue** statement outside the body of any of the listed statements.

User Action. Remove the offending **continue**.

INVCONVERT, The source or target of a conversion is noncomputational.

Error. One of the operands in the indicated line cannot be converted as specified. For example, you have attempted to cast some object to a structure.

User Action. Correct the operation.

INVDEFNAME, Missing or invalid name in **** control line; line ignored.

Warning. The indicated control line is missing a required name, as in:

```
#define
```

The entire line was ignored.

User Action. To avoid this message, correct or remove the line.

INVFILESPEC, Missing or invalid file specification in #include control line; line ignored.

Warning. The **#include** line either is missing a file or module name or specifies one that is syntactically invalid. The line was ignored.

User Action. To avoid this message and unpredictable results, correct the line.

INVFOPT, Invalid function definition option ignored.

Warning. You have specified a function definition option that is not supported. The only valid option is main__program.

User Action. To avoid this message, correct the line.

INVHEXCON, Hexadecimal constant contains an invalid character.

Error. You have specified an invalid hexadecimal constant, such as 0xG.

User Action. Correct the constant.

INVIFNAME, Missing or invalid name in #ifdef or #ifndef control line; "true" assumed.

Warning. You have specified no name, or a syntactically invalid one, in the control line; the result of the test is assumed to be true.

User Action. To avoid this message and unpredictable results, correct the line.

INVLINEFILE, Invalid file specification in #line preprocessor control line; line ignored.

Warning. The file specification is syntactically invalid, and the control line was ignored.

User Action. To avoid this message and unpredictable results, correct the line.

INVLINELINE, Missing or invalid line number in #line preprocessor control line; line ignored.

Warning. The line number is missing or is syntactically invalid, and the control line was ignored.

User Action. To avoid this message and unpredictable results, correct the line.

INVMODIDENT, Invalid ident in #module preprocessor control line; line ignored.

Warning. The ident specified in the control line either is not a valid identifier or is not a valid character-string constant.

User Action. To avoid this message and unpredictable results, correct the line.

INVMODTITLE, Missing or invalid title specification in #module preprocessor control line; line ignored.

Warning. The required title in the control line either is missing or is not a valid identifier.

User Action. To avoid this message and unpredictable results, correct the line.

INVOCTLCHAR, Invalid octal character value; high-order bits truncated.

Warning. The octal value in an escape sequence is too large, as in '\477'. Its high-order bits are truncated.

User Action. To avoid this message and unpredictable results, correct the constant.

INVPPKEYWORD, Missing or invalid keyword in preprocessor control line; line ignored.

Warning. You have written a control line with no keyword, as in:

```
# ABC
```

The line was ignored.

User Action. To avoid this message and unpredictable results, correct the line.

INVPTRADD, Invalid pointer addition; the only valid form of pointer addition is "pointer +[=] integer".

Error.

User Action. Correct the operation.

INVPTROPER, Invalid pointer arithmetic; the only math operations defined for pointers are addition and subtraction.

Error.

User Action. Correct the operation.

INVPTRSUB, Invalid pointer subtraction; the only valid forms of pointer subtraction are "pointer - pointer" or "pointer -[=] integer".

Error.

User Action. Correct the operation.

INVQUAL, One of the command qualifiers is meaningless.

Warning. One (or more) of the CC command qualifiers is meaningless to VAX-11 C, although grammatically acceptable to the VAX/VMS command interpreter (CLI).

User Action. To avoid this message and unpredictable results, correct the qualifier's name.

INVSUBS, Invalid subscript value. Subscripts must be int or unsigned. Subscripts in array declarations or casts must also be constants.

Error. You have written an invalid subscript in an array reference or declaration, or in a cast. Subscripts used in array declarations and casts must be integer constants. For example, you have put a decimal point on the constant subscript, making its type **double**.

User Action. Correct the subscript.

LIBERROR, Error while reading library "****".

Fatal. The compiler cannot read the indicated library. Either it is not a text library, or its format has been corrupted.

User Action. Verify the spelling of the library's name, and verify that it is a valid VAX/VMS text library.

LIBLOOKUP, "****" was not found in any of the specified libraries.

Fatal. The compiler failed to locate the indicated **#include** module in any of the specified or default libraries.

User Action. Check the CC command line to verify that the library containing the module was specified and that the module name, if specified, was spelled correctly. If the library was a default library, verify (with SHOW TRANSLATION C$LIBRARY) that its name is the equivalent for C$LIBRARY.

MACDEFINREF, A macro cannot be **** during the scan of a reference to the macro; line ignored.

Warning. You have tried to redefine or undefine a macro within a reference to it. The line was ignored.

User Action. To avoid this message and unpredictable results, correct the line.

MACNONTERMCHAR, Nonterminated character constant in macro argument; apostrophe added at end of line.

Warning.

User Action. To avoid this message and unpredictable results, correct the line.

MACREQARGS, Macro reference requires an argument list; "****" not substituted.

Error. You have written a macro reference without an argument list. The reference was deleted from the source file.

User Action. Correct the reference, specifying the same number of arguments as in the definition of the macro.

MACSYNTAX, Syntax error in macro definition; line ignored.

Warning.

User Action. Correct the error and recompile.

MACUNEXPEOF, Unexpected end-of-file encountered in a macro reference; "****" not substituted.

Error. The end-of-file was encountered during a macro reference; the reference was deleted.

User Action. Correct the error and recompile.

MAXMACNEST, Maximum text replacement nesting level exceeded; "****" not substituted.

Error. You have specified a macro reference which is recursive or otherwise causes repeated substitutions to a depth greater than the implementation maximum of 64.

User Action. Correct the recursion or simplify the definitions.

MISPARENS, Mismatched parentheses in #if expression; "true" expression assumed.

Warning.

User Action. To avoid this message and unpredictable results, correct the line.

MISSEDIT, Misplaced parentheses in function definition.

Error. The parentheses are either missing or misplaced in the function definition, as in:

```
double f { function-statement }
```

In the example, the function name f must be followed by a pair of parentheses, even if the function takes no parameters.

User Action. Correct the function definition.

MISSENDIF, Missing #endif preprocessor control line(s).

Error. The compiler did not encounter an **#endif** line for the most recent **#if**, **#ifdef**, or **#ifndef**.

User Action. Be sure that the control lines are properly structured, and add the missing **#endif** if appropriate.

MISSEXP, Missing or invalid exponent in float constant; zero exponent ('e0') assumed.

Warning. You have written a floating-point constant with the letter 'e' or 'E' but with no exponent or an invalid exponent. The exponent is assumed to be zero.

User Action. To avoid this message and unpredictable results, correct the constant.

MODNOMACSUB, Macro substitution cannot be performed during the scan of a macro reference; "****" not substituted; line ignored.

Warning. You have written a complex macro reference that includes a **#module** line containing a macro reference, as in:

```
macref(arg1,
#module SUBST
    .
    .
    ,arg2)
```

where SUBST has a substitution defined in a previous **#define** line. The substitution (here, for SUBST) was not performed, and the **#module** line was ignored.

User Action. To avoid this message and unpredictable results, replace the macro reference in the **#module** line with its actual value, or move the **#module** line to a position outside the complex macro reference.

MODNOTPORT, The #module preprocessor control line is not portable.

Warning. You have used the **#module** line correctly, but you are warned that it is unique to VAX-11 C and not portable.

User Action. This message occurs when the /STANDARD=PORTABLE option is used in the CC command line.

MODZERO, Constant expression includes mod zero; the result has been replaced with 0.

Warning. The constant expression has an invalid mod expression, such as 5 % 0. The result is zero.

User Action. Correct the expression (but note that its operands must not be floating-point).

NAMETOOLONG, Identifier name exceeds 31 characters; truncated to "*****".

Warning.

User Action. To avoid this message, shorten the indicated identifier.

NESTEDCOMMENT, Nested comment encountered.

Warning. You have included one comment inside another, as in /* /* comment */ */.

User Action. Check that you have not misplaced a comment delimiter and inadvertently "commented out" necessary code.

NOBJECT, No object file produced.

Informational. The compiler did not produce an object file, due to conditions reported in previous messages.

User Action. Make the corrections suggested by the other message(s).

NOCLI, This compiler can be run only from VMS DCL.

Fatal.

User Action. Recompile the program, using the CC command only with the standard command language, DCL.

NOFLOATOP, The **** operand of an **** operator may not be floating point. The operand has been converted to an integer.

Warning. The left or right operand of the indicated binary operator, or the operand of the indicated unary operator, cannot be of type **float** or **double**. It was coerced to **int**.

User Action. To avoid this message and unpredictable results, change or cast the operand to an integral type.

NOFLOATSTATE, A floating-point value has been used incorrectly in a **** statement; it has been converted to int.

Warning. A floating-point value is not valid as used in the indicated statement. It was coerced to **int**.

User Action. To avoid this message and unpredictable results, change or cast the operand to an integral type.

NOFORMALS, The declaration/definition of "****" specifies one or more function formal parameters which have been ignored.

Warning. You have included a function's formal parameters in a function declaration or definition. For example, the following function declaration is not allowed because it names the function's parameters:

```
int funct(a,b,c);
```

The parameters a, b, and c are ignored.

Similarly, the following example defines a function returning a pointer to a function returning an integer. The names of the parameters of the function returning an integer are not allowed:

```
(*f(p1,p2))(q1,q2)
int p1, p2;
{ ... }
```

User Action. To avoid this message, remove the parameters, as in:

```
int funct();
```

and

```
(*f(p1,p2))()
```

NOLABEL, Label "****" undefined in this function.

Error. You have written "**goto** label-name" for an undefined label. The scope of a label name is restricted to the function in which it is used as a label, and **goto** statements cannot branch to labels inside other functions.

User Action. If appropriate, define the appropriate label name by labeling a statement in the same function as the **goto**.

NOLISTING, No listing file produced.

Informational. The compiler did not create a listing file, usually due to previously reported errors.

User Action. None.

NONOCTALDIGIT, Octal escape sequence in a character or string constant terminated by nonoctal digit.

Warning. There is an 8 or 9 in the second or third position of an octal escape sequence. In this case, the digits preceding the nonoctal digit are evaluated, and the 8 or 9 is considered a separate character.

User Action. This message is issued only if the /STANDARD=PORTABLE option is used in the CC command line. Make sure that the compiler has resolved the ambiguity correctly.

NONOCTALESC, Escape sequence in a character or string constant starts with a nonoctal digit.

Warning. The first of three digits of an escape sequence is an 8 or 9. In this case, the backslash is ignored, and the 8 or 9 is treated as a character.

User Action. This message is issued only if the /STANDARD=PORTABLE option is used in the CC command line. Make sure that the compiler has resolved the ambiguity correctly.

NONPORADDR, Ampersand with constant is not a portable operation.

Warning. You have used an ampersand operator with a constant in the argument list of a function call. VAX-11 C permits this special case, but you are warned that the use of the ampersand on anything but lvalues is not portable.

User Action. You can suppress the message with the CC command qualifier /STANDARD=NOPORTABLE, which is the default.

NONPORTCONST, **** is a nonportable character constant.

Warning. VAX-11 C allows up to four characters to be specified in a character constant; constants of more than one character, however, are not portable.

User Action. You can suppress the message with the CC command qualifier /STANDARD=NOPORTABLE, which is the default.

NONPORTSC, **** is a nonportable storage class specifier.

Warning. This message is issued for the use of **globalref**, **globaldef**, **globalvalue**, and **readonly** storage class specifiers when the /STANDARD=PORTABLE option is used in the CC command line.

User Action. No action is required if the program need not be compatible with other C compilers. You can avoid this message by not specifying /STANDARD=PORTABLE.

NONSEQUITUR, "****" is not a member of the specified structure or union.

Warning. You have used a member name in a reference to a structure or union in which it was not declared. The reference was valid, because the member name is unique and refers unambiguously to a location in the referenced structure. This use of member names is maintained only for compatibility with older programs.

User Action. Declare the member name properly or use the /NOWARN option in the command line to suppress this message.

NONTERMCHAR, Nonterminated character constant; **** assumed.

Warning. The end of a source line was encountered before the end of a character constant (') was encountered. The indicated value was assumed.

User Action. To avoid this message and unpredictable results, correct the line.

NONTERMNULCHAR, Nonterminated character constant contains no characters; '\0' assumed.

Warning.

User Action. To avoid this message and unpredictable results, correct the constant.

NONTERMSTR, Nonterminated string constant; quotes added at end of line.

Warning.

User Action. To avoid this message and unpredictable results, correct the constant.

NOOPTIMIZATION, Complex control flow caused optimization to be suppressed for procedure or function "*****".

Informational. Optimization was not performed for the indicated function.

User Action. None.

NOSTRDEF, "*****" is a structure or union type that is not fully defined at this point in the compilation.

Warning. You have used a structure or union tag at a point where its associated type is undefined, as in:

```
struct a a_var;
.
.
struct a
     {
     int i;
     float f;
     };
```

User Action. This usage is permitted, but not recommended. To suppress the message, you must fully define a structure or union before you refer to its tag.

NOTDECL, "*****" is not declared within the scope of this usage.

Error. You have referred to an undeclared variable. All C variables must be declared explicitly; there are no defaults.

User Action. Add an appropriate declaration for the referenced object.

NOTENUM, "*****" is not an enum tag in this context.

Warning. You have used an **enum** tag before the associated type has been fully enumerated, as in:

```
enum color c1;
.
.
enum color { red, yellow };
```

User Action. This usage is permitted, but not recommended. To avoid the message, you must fully enumerate a type before you use its tag.

NOTFUNC, Function-valued expression not found.

Error. An identifier is followed by parentheses, but is neither a function name nor a dereferenced pointer to a function (*fp).

User Action. Correct the expression.

NOTINTVAL, An integer value was not found where expected.

Error. An integer value was not used where required.

User Action. Check the indicated line carefully for missing array subscripts or for other references that require integer values.

NOTLVALUE, The target of a **** operator is not an lvalue.

Error. The indicated operator, such as = (assignment) or & (address of), requires an lvalue for its operand, that is, an expression that could appear as the left operand in an assignment.

User Action. Replace the operand with a valid lvalue.

NOTPTRVAL, Address-valued expression not found.

Error. The indicated line requires a unary address (*) operator with a nonpointer operand, or a nonpointer to the left of an arrow (->) operator.

User Action. Correct the indicated line.

NOTSWITCH, Default labels and case labels valid only in switch statements.

Error. You have used **case** or **default** as a label outside the body of a **switch** statement.

User Action. Change the offending label(s).

NOTUNIQUE, "****" is not a unique member name in this context.

Error. The name of a structure or union member was used in more than one structure or union, refers to different locations in its defining structures, and was used to refer to a structure in which it was not defined. Thus, the name refers ambiguously to a place in the referenced structure.

User Action. Refer to a structure or union only with a member name declared in it.

NOWORK, No source file found in command line.

Fatal. You specified a CC command without a source file.

User Action. Recompile.

NULCHARCON, Character constant contains no characters; '\0' assumed.

Warning. You have used " for an ASCII NUL character instead of '\0'.

User Action. This usage is permitted but unconventional. Use '\0'.

NULHEXCON, Hexadecimal constant contains no digits; 0X0 assumed.

Warning. You have specified a constant such as 0X or 0x.

User Action. Be sure that 0 is a valid value in this context; if so, change the constant to 0x0.

PARMNOTUSED, Macro parameter "****" is not referenced in the definition.

Warning. A macro definition has more parameters than appear in its substitution, as in:

```
#define m(a,b,c) a*b
```

User Action. This is a possible programming error. Specify the extra parameter in the substitution or, if it is actually superfluous, delete it from the parameter list.

PPUNEXPEOF, Unexpected end-of-file encountered in preprocessor control line; line ignored.

Warning.

User Action. Examine the file to see whether the control line is necessary; if so, correct the error and recompile.

PTRASSIGN, Assignment to/from pointers and integers is nonportable.

Warning. You have assigned an integer to a pointer or an address to an integer variable. This message is issued only if /STANDARD=PORTABLE was specified.

User Action. This usage is not portable and is not recommended. The only portable assignment is pointer = 0. Change the operands or cast them to the same type.

PTRCOMPARE, Pointer comparison with nonzero integer constant or integer is a nonportable construct.

Warning. You have compared a pointer's value with something besides the constant 0. This message is issued only if /STANDARD=PORTABLE was specified.

User Action. This usage is not portable and is not recommended. The only portable comparison is of a pointer variable with 0. Otherwise, avoid the message by changing the operands or casting them to the same type.

PTRFLOATCVT, The operand of a pointer addition or subtraction operator was forced from floating-point to integer.

Warning. You have combined a pointer operand with a floating-point value, as in:

```
int i,*ip;
.
.
i = ip + 2.;
```

User Action. To avoid the message, be sure that pointers are used only with other pointers or with integers; in the above example and in similar situations, remove the decimal point from the literal constant.

REPOVERFLOW, Length of replacement text exceeds maximum buffer capacity; "****" not substituted.

Error. The length of the replacement text for a macro reference or the length of the text plus the rest of the line exceeds the implementation's limit.

User Action. Shorten the replacement text or use multiple substitutions to achieve the desired result.

SEMICOLONADDED, Semicolon added at the end of the previous source line.

Warning. A missing semicolon was added to the line prior to the line numbered in this message.

User Action. Check the previous line carefully and add the semicolon in the appropriate place.

SYMTABOVFL, The total number of symbol table pages exceeds the implementation's limit.

Fatal. The program is too complex.

User Action. Simplify the program by reducing the number and size of variables and other names, constants, and so forth.

SYNTAXERROR, ********
SYNTAXERROR2, ********
SYNTAXERROR3, ********
SYNTAXERROR4, ********

Error. The illustrated syntax error prevents the generation of an object file. (There are no important differences between the conditions causing the various forms of the message.)

User Action. Correct the errors shown.

TBLOVRFLW, Internal table overflow, too many procedures, external symbols (psects), or the program is too complex.

Fatal. Either the source file contains too many functions or expressions, or the compiler has overflowed its virtual address space.

User Action. Reduce the size of the source file by dividing it into smaller, separately compilable files, or change the logic of the program to reduce the number of complicated expressions.

TOOFEWARGS, Argument list for macro "****" contains too few arguments; missing arguments assumed to be null.

Warning. You have written a reference to the indicated macro with fewer arguments than are specified in its definition.

User Action. To avoid this message and unpredictable results, correct the reference.

TOOMANYARGS, Argument list for macro "****" contains too many arguments; excess arguments ignored.

Warning. You have written a reference to the indicated macro with more arguments than are specified in its definition.

User Action. To avoid this message and unpredictable results, shorten the argument list.

TOOMANYCHAR, Character constant contains too many characters; truncated to ****.

Warning. The length of a character constant exceeds the implementation limit (four characters). The constant was truncated to the indicated value.

User Action. Reduce the length of the indicated character constant.

TOOMANYERR, The total number of errors exceeds the implementation's limit of 100.

Fatal.

User Action. Correct the errors reported in previous compiler messages and recompile.

TOOMANYINITS, The initializer list for "****" specifies too many initializers; excess initializers ignored.

Warning. This message can be issued for any type of variable. Some causes might be missing brackets from an array declaration, misplaced braces in an initializer list that would cause array elements to be skipped, or simply, more initializers than elements in an array or structure.

User Action. Check the declaration and make the appropriate corrections.

TOOMANYPARM, Too many macro parameters; excess parameters ignored.

Warning. The number of macro parameters in a **#define** line exceeds the implementation limit of 64.

User Action. Reduce the number of parameters.

TOOMANYSTR, String constant contains too many characters; truncated.

Warning. The character-string constant in this line exceeds the implementation's limit of 1000 characters.

User Action. Shorten the constant.

TRUNCFLT, Double-precision floating-point constant cannot be converted to single precision; 0.0 assumed.

Warning.

User Action. Ensure that 0 is a valid value in this context; if necessary, redeclare the conversion target as **double**.

TRUNCSTRINIT, String initialization for "****" contains too many characters to fit; truncated.

Warning. If the variable is a simple one-dimensional array, the initializer is truncated (such that the length of the initializer is array–1) and the null byte is added to the end of the array. If the array is a multidimensional array or an array within a structure, the initializer is truncated to the length of the array and a null byte is not added.

User Action. You should treat arrays of characters as strings, that is, allowing for the null byte at the end of the array. The special case of multidimensional arrays and arrays within structures should be taken into account, especially when you do not want the null byte to be truncated.

UABORT, Compilation terminated by user.

Fatal. The compilation was terminated by a DCL CTRL/C command.

User Action. None.

UNDEFIFMAC, "****" is not a currently defined macro; constant zero assumed.

Warning. The identifier in a constant expression in an **#if** preprocessor control line is not currently defined as a macro. The expression is evaluated as if the identifier were a constant 0.

User Action. Define the identifier as a macro or remove the reference to it.

UNDEFNAME, "*****" is already undefined; line ignored.

Warning. The specified identifier (in an **#undef** line) was either never defined or else occurred in a previous **#undef**.

User Action. Remove the **#undef**, or, if applicable, add the definition of the identifier at the appropriate place.

UNEXPCTL, Unexpected **** control line encountered; line ignored.

Warning. The specified control line occurred out of place and was ignored.

User Action. Check the logic of all control lines in the program to be sure that it is valid.

UNEXPEND, Unexpected end-of-**** encountered in #define control line; line ignored.

Warning. The end of the **#define** line or end of the source file was encountered before the definition was complete.

User Action. To avoid this message and unpredictable results, correct the line.

UNEXPEOF, Unexpected end-of-file encountered in a ****.

Error. The unexpected end-of-file prevents the generation of code.

User Action. Correct the error and recompile.

UNRECCHAR, Unrecognized character ignored.

Warning. The line contains either an entirely meaningless character or one that appears out of its proper context, for example, a number sign (#) that is not the first character on a line.

User Action. Move or remove the character.

WARNSUM, Completed with warnings.

Warning. The compilation was completed, but several warning messages were issued. The program may not be logically correct.

User Action. If the execution of the program does not produce the expected results, correct the statements for which warnings were issued.

Appendix D

Compiler Listing Formats

The VAX-11 C compiler has many options that let you control what information is included in the listing file. This appendix shows the different listing formats that are available.

When the CC command line contains the /LIST qualifier but does not contain the /SHOW qualifier, you are given the default listing. All the information in the default listing is also included in the other listing formats. The default listing includes:

- Margin information
- The C source text
- Any errors encountered during the compilation
- The command line used to invoke the compiler

The left-hand margin of the source listing produced by the VAX-11 C compiler contains several items of information, arranged into fields in the following format:

nnnnn i ss mm

nnnnn is the compiler-generated listing line number; it starts at 1, and is incremented by one for each line in the source program, including lines read from included files (whether or not the /SHOW=INCLUDE qualifier was specified in the command line).

i is the level of nesting of lines read from included files; this field is present only if /SHOW=INCLUDE was specified on the command line. Level 0, which appears as a blank, indicates lines read from the source file(s) specified on the command line.

ss is the level of nesting of compound statements in the source program; it starts at zero (which appears as a blank) for external definitions and declarations, is incremented each time a left brace which introduces a compound statement is encountered (including the brace which introduces a function body), and is decremented at the corresponding

right brace. This field may also appear as an "X" instead of a number; this indicates that the source line is being ignored by the compiler as a result of the evaluation of a previous **#if**, **#ifdef**, or **#ifndef** preprocessor control line.

mm — is the level of nesting of the last macro expanded in the line; this field is present only if the qualifier /SHOW=EXPANSIONS or /SHOW=INTERMEDIATE was specified on the command line. Level 0 corresponds to the original source line, and appears as a blank. When this field is nonzero, however, the fields "nnnnn", "i", and "ss" all appear as blanks.

In all cases, the numbers listed are right-justified in their fields, with no leading zeros.

```
❶
EXAMPLE
V1.0                                    ❷ 30-OCT-1981 15:17:19    VAX-11 C    V1.0-00              ❹ Page  1
                                        ❸ 23-OCT-1981 14:01:11    _DBA5:[CPROG]EXAMPLE.C;8               (1)

     ❺
      1          /* This is a sample program to show the format of the compiler listing,
      2             as well as the effect of the various /SHOW command-line qualifier values */
      3
      4        ❻ /* This line is here to show what happens when a line from the source file is so long that it extends beyond the rig
      5
      6          #include "debugging.h"
     10
     11          #include timeb
     29
     30          #define NULL    0
     31
     32          main ()
     33     ❼    {
❽    34     1            globalvalue     SS$_NORMAL;
%CC-W-NONPORTSC, globalvalue is a nonportable storage class specifier.

     35     1            struct  timeb   time_struct;
     36     1            char    *ctime_string, *ctime();
     37     1            int     status = SS$_NORMAL;
     38     1
     39     1            ftime (& time_struct);
     40     1            if ((ctime_string = ctime (& time_struct.time)) != NULL)
     41     1                    printf ("Run time is %s", ctime_string);
     42     1
     43     1      #if debugging
     44     X            printf ("\n*** Debugging version ***\n\n");
     45     X            status =
     46     1      #endif
     47     1              process ();
     48     1
     49     1            return status;
     50     1      }


❾
Command Line
------------

/LIST/STANDARD=PORTABLE EXAMPLE
```

Figure D–1: Default Compiler Listing

Figure D–1 shows the default compiler listing. The /STANDARD=PORTABLE option was used to show how the compiler lists error messages. The notes to the figure are keyed to the numbers appearing on the listing.

❶ The name of the module and its identification appear at the top left of the listing.

❷ The date and time of compilation and the version of the compiler that was used appear at the center of the listing.

❸ The date and time when the source file was created and the file specification appear below the information in number ❷.

❹ The page numbers of the listing file and the source file (in parentheses) are shown to the far right of the listing.

❺ The listing line numbers are generated by the compiler.

❻ Long source lines are truncated if neither /SHOW=EXPANSION nor /SHOW=INTERMEDIATE is specified.

❼ The nesting level of compound statements starts at zero (appearing as blanks), is incremented each time a left brace is encountered, or appears as an "X", which means that the line is ignored after a conditional control line is evaluated.

❽ Diagnostic messages appear immediately following the source line in question.

❾ The command line that generated the listing appears at the bottom.

The CC command line qualifiers /SHOW=EXPANSIONS and /SHOW=INTERMEDIATE cause the compiler listing to display the results of macro substitution. /SHOW=EXPANSIONS displays only the final, fully-substituted line, immediately following the listing of the original source line. /SHOW=INTERMEDIATE displays the complete progress of substitution, printing a new line each time a macro reference is replaced by its definition. If an error message is issued against a line which contains substitutions, the message appears between the original line and the first substituted line.

The purpose of displaying the results of macro substitution is to show the source line as it is ultimately seen by the compiler. Since macro substitution may significantly increase the length of the source line, the specification of either of the above qualifiers also causes the compiler to "wrap" any source line that would exceed the right-hand listing margin

(whether or not the line contained any substitutions). The wrapped portion of the line appears on a new listing line, with all of the left-margin fields appearing as blanks. Note that if neither of these qualifiers is specified, any source lines which exceed the right-hand margin are simply truncated.

Figure D–2 is a listing which shows **#include** modules and intermediate macro expansions.

❶ The long line is wrapped around when the /SHOW=INTERMEDIATE or /SHOW=EXPANSION qualifier is used.

❷ The level of nesting of lines read from **#include** files is shown. This column is blank when the lines are read from the source file.

❸ The nesting level is incremented when the **#include** file also contains **#include** control lines.

❹ The level of nesting of the last macro expanded in the line is shown.

```
EXAMPLE                                          30-OCT-1981 14:18:46    VAX-11 C    V1.0-00                Page 1
V1.0                                             23-OCT-1981 14:01:11    _DBA5:[CPROG]EXAMPLE.C;8                (1)


     1          /* This is a sample program to show the format of the compiler listing,
     2             as well as the effect of the various /SHOW command-line qualifier values */
     3
     4        ❶ /* This line is here to show what happens when a line from the source file is so long that it extends beyond the
                   right-hand listing margin */
     5
     6 ❷      #include "debugging.h"
     7  1       #define YES     1
     8  1       #define NO      0
     9  1       #define debugging       NO      /* YES only if debugging code is to be compiled */
    10
```

```
11                  #include timeb
12 1                /*
13 1                 *       timeb - ftime system call return definition include file
14 1                 */
15 1
16 1                #include types
17 2 ❸              /*
18 2                 *       types - type definitions include file
19 2                 */
20 2                typedef long int time_t;
21 1
22 1
23 1                struct timeb {
24 1                        time_t  time;
25 1                        unsigned short millitm;
26 1                        short   timezone;
27 1                        short   dstflag;
28 1                        };
29
30                  #define NULL    0
31
32                  main ()
33                  {
34    1                     globalvalue     SS$_NORMAL;
35    1                     struct  timeb   time_struct;
36    1                     char    *ctime_string, *ctime();
37    1                     int     status = SS$_NORMAL;
38    1
39    1                     ftime (& time_struct);
40    1 ❹                   if ((ctime_string = ctime (& time_struct.time)) != NULL)
      1                     if ((ctime_string = ctime (& time_struct.time)) != 0)
41    1                             printf ("Run time is %s", ctime_string);
42    1
43    1               #if debugging
         1            #if NO
         2            #if 0
44    X                     printf ("\n*** Debugging version ***\n\n");
45    X                     status =
46    1               #endif
47    1                       process ();
48    1
49    1                     return status;
50    1               }
```

Figure D–2: Listing Format of Macro Substitutions

When the /CROSS_REFERENCE option is used, the compiler produces a storage map with symbol table cross-references, as shown in Figure D–3. The storage map produced by the /SHOW=SYMBOLS is the same as the listing shown in Figure D–3, except that the cross-references are not included.

❶ The "External Declarations" section of the Storage Map lists all externally declared names (that is, names declared or defined outside of any function).

❷ When the /CROSS_REFERENCE option is used, the compiler gives the line number in which each name is referenced. The cross-reference information is not included in the storage map unless the /CROSS_REFERENCE option is used.

❸ For each function in the source program, the compiler lists each declared name, giving:
- The identifier of the name
- The line on which the name is declared
- The size of the identifier
- The storage class to which the name belongs
- The data type of the name

❹ The Program Section (Psect) Synopsis lists the program sections created by the compiler and their attributes.

❺ The Function Definition Map lists each function defined in the program and gives the line number in which the function is defined.

```
EXAMPLE                                  30-OCT-1981 14:18:47    VAX-11 C    V1.0-00              Page  2
V1.0                                     23-OCT-1981 14:01:11    _DBA5:[CPROG]EXAMPLE.C;8             (1)

                                    +-------------+
                                    ! Storage Map !
                                    +-------------+


❶
External Declarations
---------------------

 Identifier Name                Line    Size            Class
 ---------------                ----    ----            -----

main                            32                      Extern def.

timeb                           23      10 bytes
                                                                ❷
 time                           24      1 longword

 millitm                        25      1 word

 timezone                       26      1 word

 dstflag                        27      1 word

time_t                          20      1 longword
```

Figure D–3: Cross-Reference Listing

```
Function "main" defined at line 32
----------------------------------
❸
 Identifier Name                Line    Size            Class
 ---------------                ----    ----            -----

ctime                           36                      Extern

ctime_string                    36      1 longword      Register

ftime                           39                      Extern

printf                          41                      Extern

process                         47                      Extern

SS$_NORMAL                      34      1 longword      Globalvalue

status                          37      1 longword      Register

time_struct                     35      10 bytes        Auto

 time                           24      1 longword

 millitm                        25      1 word

 timezone                       26      1 word

 dstflag                        27      1 word


❹
Psect Synopsis
--------------

 Psect Name                     Allocation
 ----------                     ----------

$CODE                           68 bytes
$CHAR_STRING_CONSTANTS          15 bytes
```

```
❺
Function Definition Map
-----------------------

 Line    Name
 ----    ----

32       main


Command Line
------------

/SHOW=(INCLUDE,INTERMEDIATE)/CROSS_REFERENCE EXAMPLE



Type and References
-------------------

Function returning long int
- No references
Structure tag
- Referenced at line 35
 Member (offset = 0), long int
 - Referenced at line 40
 Member (offset = 4 bytes), unsigned short int
 - No references
 Member (offset = 6 bytes), short int
 - No references
 Member (offset = 8 bytes), short int
 - No references
Typedef:  long int
- Referenced at line 24
```

Figure D–3: (Cont.) Cross-Reference Listing

```
Type and References
-------------------

Function returning pointer to char
- Referenced at line 40
Pointer to char
- Referenced at lines 40 and 41
Function returning long int
- Referenced at line 39
Function returning long int
- Referenced at line 41
Function returning long int
- Referenced at line 47
Long int
- Referenced at line 37
Initialized long int
- Referenced at line 49
Struct timeb
- Referenced at lines 39 and 40
 Member (offset = 0), long int
 - Referenced at line 40
 Member (offset = 4 bytes), unsigned short int
 - No references
 Member (offset = 6 bytes), short int
 - No references
 Member (offset = 8 bytes), short int
 - No references



Attributes
----------

Position-independent, relocatable, shareable, executable, readable
Position-independent, relocatable, readable, writeable
```

Figure D–3: (Cont.) Cross-Reference Listing

When the /SHOW=STATISTICS option is used, the compiler accumulates and displays statistics for each phase of its operation. It then lists the amount of I/O, memory, and CPU time used during the compilation. Figure D–4 shows the information returned by the /SHOW=STATISTICS option.

❶ This column shows the maximum working set size, not the total.

❷ The subphases of the compiler are indented in this column and precede the totals for their phase.

❸ The phase totals follow the breakdown of their subphases.

❹ The totals for the performance indicators may be greater than the sum of the phase totals. For example, the buffered I/O total (buf I/O) is 11, not 10.

❺ The compilation rate is the number of lines compiled per minute of CPU time. CPU times are measured in 10-millisecond units.

```
EXAMPLE                                  30-OCT-1981 09:44:32    VAX-11 C    V1.0-00              Page 2
V1.0                                     23-OCT-1981 14:01:11    _DBA5:[CPROG]EXAMPLE.C;8             (1)

                                    +-------------------------+
                                    ! Performance Indicators !
                                    +-------------------------+

  Phase                        buf i/o dir i/o pageflt virtmem workset❶ cputim
  -----                        ------- ------- ------- ------- -------  ------

  Parse/semantics totals             9       7     331       0     362      90
❷   live analysis                    0       0      18       0     362       6
    reorder invariants               0       0       7       0     362       3
    eliminate redundancy             0       0       2       0     362       2
    eliminate assignments            0       0       0       0     362       0
❸ optimizer totals                   0       0      40       0     362      14
  allocator totals                   0       0      15       0     362       1
    generate code list               0       0      32       0     362       7
    register allocation              0       0       0       0     362       1
    peephole optimization            0       0       3       0     362       1
    branch/jump resolution           0       0       0       0     362       0
    write object module              0       0       7       0     362       1
  code generator totals              1       1      68       0     362      16
❹ total compilation                 11       8     477      16     362     128

  50 lines compiled
  compilation rate was 2343 lines per minute❺
```

Figure D–4: Compiler Performance Statistics

Finally, when you use the /MACHINE_CODE option, a listing file is created showing the assembly language and machine code generated by the compiler. This information is generated in line with the C source statements and is listed below the last substituted line.

Figure D–5 shows the listing generated by the /MACHINE_CODE option. The notes that follow the figure are keyed to the numbers in the figure.

```
EXAMPLE
V1.0                                                    30-OCT-1981 09:46:26    VAX-11 C    V1.0-00              Page 1
                                                        23-OCT-1981 14:01:11    _DBA5:[CPROG]EXAMPLE.C;8             (1)

     1          /* This is a sample program to show the format of the compiler listing,
     2             as well as the effect of the various /SHOW command-line qualifier values */
     3
     4          /* This line is here to show what happens when a line from the source file is so long that it extends beyond the rig
     5
     6        ❶ #include "debussing.h"                                                    ANTS
    10
    11          #include timeb
    29
    30          #define NULL    0
    31
    32          main ()
    33          {                           ❷               ❸
                                          0000  main:
                                    0000  0000            .entry  main,^m<>
                                5E 0C C2  0002            subl2   #C,sp
                           00000000* EF 16  0005            jsb     C$MAIN

    34     1              globalvalue     SS$_NORMAL;
    35     1              struct  timeb   time_struct;
    36     1              char    *ctime_string, *ctime();
    37     1              int     status = SS$_NORMAL;
                        5C 00000000* 8F D0  0008            movl    #SS$_NORMAL,ap

    38     1
    39     1              ftime (& time_struct);
                                FB AD 9F    0012            pushab  -0A(fp)
                           00000000* EF 01 FB   0015        calls   #1,ftime

    40     1              if ((ctime_string = ctime (& time_struct.time)) != NULL)
                                FB AD DF    001C            pushal  -0A(fp)
                           00000000* EF 01 FB   001F        calls   #1,ctime
                                   50 D5    0026            tstl    r0
                                   0F 13    0028            beql    vcg.1
```

Figure D–5: Machine Code Listing

```
41    1                     printf ("Run time is %s", ctime_string);
                              50 DD     002A            pushl   r0
                        00000000 EF DF     002C            pushal  $CHAR_STRING_CONST
                    00000000* EF 02 FB     0032            calls   #2,printf

42    1
43    1       #if debugging
44    X               printf ("\n*** Debugging version ***\n\n");
45    X               status =
46    1       #endif
47    1                 process ();
                                           0039  vcg.1:
                    00000000* EF 00 FB     0039            calls   #0,process

48    1
49    1               return status;
                              50 5C D0     0040            movl    ap,r0
                                 04     0043            ret

50    1       }

Command Line
------------

/MACHINE_CODE EXAMPLE
```

Figure D–5: (Cont.) Machine Code Listing

❶ The C source statements are shown.

❷ The object module location of each statement and the machine code instructions are listed.

❸ The assembly language code generated by each line of source text is shown beside its corresponding machine code instruction.

Appendix E

VAX-11 C Definition Modules

Table E-1 lists the definition modules contained in the text library named SYS$LIBRARY:CSYSDEF.TLB. The exact content of the modules is not included here because the modules may change from release to release.

The contents of these modules can be examined in the appropriate definition file. All definition files have the file type H and they are contained in SYS$LIBRARY. You can print or type individual files, or you can issue the following command to print all the files with their file names appearing at the top of each page:

```
$ PRINT SYS$LIBRARY:*.H/HEADER
```

Table E-1: VAX-11 C Definition Modules

accdef	Accounting file record definitions
chfdef	Structure definitions for condition handlers
ctype	Character type and macro definitions for character classification functions
dcdef	Device class and type code definitions
descrip	Descriptor structure and constant definitions
errno	Error number definitions
errnodef	VAX-11 C error message constants
fab	File access block definitions
iodef	I/O function code definitions
jpidef	$GETJPI system service request code definitions
math	Math function definitions
nam	Name block definitions
opcdef	OPCOM request code definitions
pqldef	Process quota code definitions
prvdef	Privilege mask bit definitions
psldef	Processor status longword definitions

Table E-1: (Cont.) VAX-11 C Definition Modules

rab	Record access block definitions
rms	All RMS structures and return status value definitions
rmsdef	RMS return status value definitions
secdef	Image section flag bit and match constant definitions
setjmp	**setjmp** and **longjmp** state buffer definition
sfdef	Stack call frame definitions
signal	Signal value definitions
ssdef	System service return status value definitions
stdio	Standard I/O definitions
time	**localtime** definitions
timeb	**ftime** definitions
ttdef	Terminal definitions
types	Type definitions
xab	Extended attribute block definitions

As noted in Chapter 6, the **errno** external variable is useful in determining the cause of a run-time error. When an error occurs during a function call, the function returns an unsuccessful status to the program and sets **errno** to a value that indicates the reason for the failure.

The errno definition module declares the **errno** variable and symbolically defines the possible **errno** values. By including the errno definition module in your program, you can check for specific values after a function call. These values, and their meanings, are as listed in Table E-2.

Table E-2: errno Symbolic Values

Symbolic Constant	Description
EPERM	Not owner
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Argument list too long
ENOEXEC	exec format error
EBADF	Bad file number
ECHILD	No child processes
EAGAIN	No more processes
ENOMEM	Not enough memory

Table E-2: (Cont.) errno Symbolic Values

Symbolic Constant	Description
EACCES	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required
EBUSY	Mount device busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	File table overflow
EMFILE	Too many open files
ENOTTY	Not a typewriter
ETXTBSY	Text file busy
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
EVMSERR	VMS-specific error code for nontranslatable errors

The **errno** values can also be translated into a UNIX-like message by the **perror** function. If **perror** cannot translate the **errno** value, it prints the following message, followed by the VAX/VMS error message associated with the value:

```
%s:non-translatable vms error code: xxxxxx vms message:
```

%s is the string you supply to **perror**; xxxxxx is the VAX/VMS message number.

Example E–1 shows the use of the errno definition module to check for a domain error during a call to the **sqrt** function; the program uses **perror** to print a UNIX-like message if the error occurs.

```
#include errno
#include math
#include stdio

main( )
{
     double input,square_root;

     printf("Enter a number: ");
     scanf("%e",&input);

     square_root = sqrt(input);

     /* CHECK FOR DOMAIN ERROR AND PRINT UNIX-LIKE
        MESSAGE IF THE ERROR OCCURS */
     if (errno == EDOM)
      perror("Example -- input was negative");
     else
      printf("square root of %e = %e\n",
            input,square_root);
}
```

Example E–1: Checking the errno Variable

Appendix F

VAX-11 C Run-Time Modules and Entry Points

This appendix summarizes the modules and entry points in the VAX-11 C run-time system. Table F-1 lists the modules in the library and describes their function. Table F-2 lists the entry points defined in each module and describes their function. Table F-3 lists the modules from the VMS Run-Time Procedure Library that are called by VAX-11 C run-time modules.

Table F-1: VAX-11 C Run-Time Modules

Module	Description
C$$CLEANUP	Flush and close all files
C$$DATA	Data definitions of standard file structures
C$$DOPRINT	Character-string print and scan routines
C$$FILBUF	Fill a file buffer
C$$FLSBUF	Flush a file buffer
C$$MAIN	Main start-off routine for C programs
C$$MATH_HAND	Math routine condition handler
C$$TRANSLATE	Translate VMS codes to UNIX codes
C$ABORT	Abort the current process
C$ABS	Integer absolute value math function
C$ACOS	Arc cosine math function
C$ALARM	Set alarm function
C$ASIN	Arc sine math function
C$ATAN	Arc tangent math function
C$ATAN2	Arc tangent math function
C$ATOF	ASCII to floating-point binary conversion
C$ATOL	ASCII to integer binary conversion
C$BREAK	Memory allocation routines
C$CEIL	Ceiling math function
C$COS	Cosine math function
C$COSH	Hyperbolic cosine math function
C$CTERMID	Controlling terminal identification
C$CTYPE	Character type data definitions
C$CUSERID	User-name identification
C$ECVT	**Double float** to ASCII string conversion

Table F-1: (Cont.) VAX-11 C Run-Time Modules

Module	Description
C$ERRNO	Run-time library error message definitions
C$EXIT	Close files and exit image
C$EXP	Base e exponentiation math function
C$FABS	Floating-point **double** absolute math function
C$FCLOSE	Close a file
C$FDOPEN	Open a file by file descriptor
C$FFLUSH	Flush a file buffer
C$FGETC	Get a character from a file
C$FGETNAME	Get a file-name string
C$FGETS	Get a string from a file
C$FLOOR	Floor math library function
C$FOPEN	Open a file
C$FPUTC	Write a character to a file
C$FPUTS	Write a string to a file
C$FREAD	Read from a file to a buffer
C$FREXP	Extract fraction and exponent math function
C$FSEEK	Position to a byte offset within a file
C$FTELL	Return current byte offset within a file
C$FWRITE	Write from a buffer to a file
C$GCVT	**double** value to ASCII string conversion
C$GETCHAR	Get a character from standard input
C$GETENV	Get environment value
C$GETGID	Get group identification
C$GETPID	Get the process identification
C$GETS	Get a string from standard input
C$GETUID	Get user identification
C$GETW	Get a longword from an input file
C$HYPOT	Euclidean distance math library function
C$KILL	Send signal to process
C$LDEXP	Power of 2 math library function
C$LOG	Logarithm base e math library function
C$LOG10	Logarithm base 10 math library function
C$MAIN	C main routines
C$MALLOC	Memory allocation
C$MODF	Extract fraction and integer math function
C$NICE	Set process priority
C$PAUSE	Suspend the process until a signal is received
C$PERROR	Print an error message
C$POW	Power math library function
C$PRINTF	Formatted output routine
C$PUTCHAR	Write a character to the standard output
C$PUTS	Write a string to the standard output
C$PUTW	Write a longword to a file
C$RAND	Random number generator
C$REWIND	Return file pointer to the beginning of the file
C$RMS_PROTOTYPES	Definition of RMS data structures

Table F-1: (Cont.) VAX-11 C Run-Time Modules

Module	Description
C$SCANF	Formatted input routine
C$SETBUF	Associate a buffer with a file
C$SETGID	Set group identification
C$SETJMP	Non-local goto functions (**setjmp/longjmp**)
C$SETUID	Set user identification
C$SIGNAL	Manipulate signal database
C$SIN	Sine math function
C$SINH	Hyperbolic sine math function
C$SLEEP	Suspend the process for a number of seconds
C$SQRT	Square root math function
C$STRCAT	String concatenation
C$STRCHR	Search for a character in a string
C$STRCMP	String comparison
C$STRCPY	String copy
C$STRCSPN	Search for a character in a set of characters
C$STRLEN	Determine the string length
C$STRNCAT	String concatenation
C$STRNCMP	String comparison
C$STRNCPY	String copy
C$STRPBRK	Search a string for a set of characters
C$STRRCHR	Search a string
C$STRSPN	Search a string for characters not in a set
C$TAN	Tangent math library function
C$TANH	Hyperbolic tangent math function
C$TIME	Get real-time values
C$TIMEF	Manipulate/convert real-time values
C$TMPFILE	Create a temporary file
C$TMPNAM	Generate a name for a temporary file
C$TOLOWER	Uppercase to lowercase conversion
C$TOUPPER	Lowercase to uppercase conversion
C$UNGETC	Push a character back into an input stream
C$UNIX	UNIX emulation routines
C$VFORK	Spawn a process

Table F-2: VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
abort	C$ABORT	Abort the current process
abs	C$ABS	Integer absolute value math library function
access	C$UNIX	Check the accessibility of a file
acos	C$ACOS	Arc cosine math library function

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
alarm	C$ALARM	Set alarm library function
asin	C$ASIN	Arc sine math library function
atan	C$ATAN	Arc tangent math library function
atan2	C$ATAN2	Arc tangent math library function
atof	C$ATOF	Convert ASCII to floating-point binary
atoi	C$ATOL	Convert ASCII to integer binary
atol	C$ATOL	Convert long ASCII to binary
brk	C$BREAK	Determine the low virtual address for program data area
c$$cleanup	C$$CLEANUP	Flush and close all files
c$$cond_hand	C$$MAIN	Image condition handler
c$$ctrlc_hand	C$$MAIN	Control/C ast handler
c$$doprint	C$$DOPRINT	Internal output formatting routine
c$$doscan	C$$DOSCAN	Internal input formatting routine
c$$endopen	C$FOPEN	Internal file opening routine
c$$environ	C$UNIX	Establish **vfork** environment
c$$exhandler	C$UNIX	Emulator exit handler
c$$filbuf	C$$FILBUF	Fill input buffer
c$$flsbuf	C$$FLSBUF	Flush a file buffer
c$$main	C$$MAIN	Main start-up routine
c$$math_hand	C$$MATH_HAND	Math condition handler
c$$translate	C$$TRANSLATE	Translate VAX/VMS error codes to UNIX error codes
c$$vfork	C$$VFORK	**vfork** start-up image
c$main	C$MAIN	Start up main program with no arguments
c$main_args	C$MAIN_ARGS	Start up main program with arguments
cabs	C$HYPOT	Euclidean distance math library function
calloc	C$MALLOC	Allocate and clear storage
cc$rms_fab	C$RMS_PROTOTYPES	File access block prototype
cc$rms_nam	C$RMS_PROTOTYPES	Name block prototype

Shaded entry points are reserved for DIGITAL use.

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
cc$rms_rab	C$RMS_PROTOTYPES	Record access block prototype
cc$rms_xaball	C$RMS_PROTOTYPES	Allocation control extended attribute block prototype
cc$rms_xabdat	C$RMS_PROTOTYPES	Date and time extended attribute block prototype
cc$rms_xabfhc	C$RMS_PROTOTYPES	File header characteristics extended attribute block prototype
cc$rms_xabkey	C$RMS_PROTOTYPES	Indexed file key extended attribute block prototype
cc$rms_xabpro	C$RMS_PROTOTYPES	File protection extended attribute block
cc$rms_xabrdt	C$RMS_PROTOTYPES	Revision date and time extended attribute block prototype
cc$rms_xabsum	C$RMS_PROTOTPYES	Summary extended attribute block prototype
ceil	C$CEIL	Ceiling math library function
cfree	C$MALLOC	Deallocate storage
chdir	C$UNIX	Change the default directory
chmod	C$UNIX	Change a file's access mode
chown	C$UNIX	Change a file's owner
close	C$UNIX	Close a file
cos	C$COS	Cosine math library function
cosh	C$COSH	Hyperbolic cosine math library function
creat	C$UNIX	Create a file
ctermid	C$CTERMID	Identify the controlling terminal
ctime	C$TIMEF	Convert time to an ASCII string
cuserid	C$CUSERID	Identify the user name
delete	C$UNIX	Delete a file by file name
dup	C$UNIX	Create a duplicate file descriptor
dup2	C$UNIX	Create a duplicate file descriptor
ecvt	C$ECVT	Convert a **double** value to ASCII
execl	C$UNIX	Execute a program image
execle	C$UNIX	Execute a program image

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
execv	C$UNIX	Execute a program image
execve	C$UNIX	Execute a program image
exit	C$EXIT	Close files and exit
_exit	C$UNIX	Exit image
exp	C$EXP	Base e exponentiation math function
fabs	C$FABS	**double** absolute math function
fclose	C$FCLOSE	Close a file
fcvt	C$ECVT	Convert a **double** value to ASCII
fdopen	C$FDOPEN	Open a file by file descriptor
fflush	C$FFLUSH	Flush a file buffer
fgetc	C$FGETC	Get a character from a file
fgetname	C$FGETNAME	Get a file-name string
fgets	C$FGETS	Get a string from a file
floor	C$FLOOR	Floor math library function
fopen	C$FOPEN	Open a file by file pointer
fprintf	C$PRINTF	Format a string to a file
fputc	C$FPUTC	Write a character to a file
fputs	C$FPUTS	Write a string to a file
fread	C$FREAD	Read from a file
free	C$MALLOC	Deallocate storage
freopen	C$FOPEN	Close and reopen a file
frexp	C$FREXP	Extract fraction exponent math function
fscanf	C$SCANF	Scan input from a file
fseek	C$FSEEK	Position to an offset in a file
ftell	C$FTELL	Return current offset in a file
ftime	C$TIME	Get the time
fwrite	C$FWRITE	Write to a file
gcvt	C$GCVT	Convert a **double** value to ASCII
getchar	C$GETCHAR	Get a character from standard input
getegid	C$GETGID	Get the effective group identification
getenv	C$GETENV	Get an environment value
geteuid	C$GETUID	Get the effective user identification
getgid	C$GETGID	Get the group identification
getname	C$UNIX	Get a file-name string
getpid	C$GETPID	Get the process identification

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
gets	C$GETS	Get a string from standard input
getuid	C$GETUID	Get the user identification
getw	C$GETW	Get a longword from an input file
gsignal	C$SIGNAL	Generate a signal
hypot	C$HYPOT	Euclidean distance math library function
isatty	C$UNIX	Check for a terminal file
kill	C$KILL	Send a signal to a process
ldexp	C$LDEXP	Power of 2 math library function
localtime	C$TIMEF	Place time in a time structure
log	C$LOG	Logarithm base e math library function
log10	C$LOG10	Logarithm base 10 math library function
longjmp	C$SETJMP	Return to **setjmp**'s entry point
lseek	C$UNIX	Seek to a position in a file
malloc	C$MALLOC	Allocate memory
mktemp	C$TMPNAM	Make a temporary file-name string
modf	C$MODF	Extract fraction and integer math function
nice	C$NICE	Set process priority
open	C$UNIX	Open a file by file descriptor
pause	C$PAUSE	Suspend the process
perror	C$PERROR	Print an error message
pipe	C$UNIX	Allow two processes to exchange data
pow	C$POW	Power math library function
printf	C$PRINTF	Format a string to standard output
putchar	C$PUTCHAR	Write a character to standard output
puts	C$PUTS	Write a string to standard output
putw	C$PUTW	Write a longword to a file
rand	C$RAND	Compute a random number
read	C$UNIX	Read a file
realloc	C$MALLOC	Change the size of an area of storage

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
rewind	C$REWIND	Return to the beginning of the file
sbrk	C$BREAK	Add bytes to the program's low virtual address
scanf	C$SCANF	Format input from the standard input
setbuf	C$SETBUF	Associate a buffer with a file
setgid	C$SETGID	Set group identification
setjmp	C$SETJMP	Set up a return site for **longjmp**
setuid	C$SETUID	Set user identification
signal	C$SIGNAL	Set a signal
sin	C$SIN	Sine math library function
sinh	C$SINH	Hyperbolic sine math library function
sleep	C$SLEEP	Suspend the process
sprintf	C$PRINTF	Format a string to a memory buffer
sqrt	C$SQRT	Square root math library function
srand	C$RAND	Reinitialize the random number generator
sscanf	C$SCANF	Format input from memory
ssignal	C$SIGNAL	Set a signal
strcat	C$STRCAT	Concatenate two strings
strchr	C$STRCHR	Search for a character in a string
strcmp	C$STRCMP	Compare two strings
strcpy	C$STRCPY	Copy a string to another string
strcspn	C$STRCSPN	Search string for a character
strlen	C$STRLEN	Determine the length of a string
strncat	C$STRNCAT	Concatenate two strings
strncmp	C$STRNCMP	Compare two strings
strncpy	C$STRNCPY	Copy from one string to another
strpbrk	C$STRPBRK	Search a string for a character
strrchr	C$STRRCHR	Search a string for a character
strspn	C$STRSPN	Search a string for a character
tan	C$TAN	Tangent math library function

Table F-2: (Cont.) VAX-11 C Run-Time Entry Points

Entry Point	Module	Description
tanh	C$TANH	Hyperbolic tangent math library function
time	C$TIME	Get the epoch time
times	C$UNIX	Get the process and CPU times
tmpfile	C$TMPFILE	Create a temporary file
tmpnam	C$TMPNAM	Generate a temporary file name
tolower	C$TOLOWER	Convert uppercase to lowercase
toupper	C$TOUPPER	Convert lowercase to uppercase
umask	C$UNIX	Set a file's protection mask
ungetc	C$UNGETC	Push a character back into the stream
vfork	C$UNIX	Spawn a process
wait	C$UNIX	Suspend a process
write	C$UNIX	Write a file

Table F-3: Run-Time Library Procedures Called by VAX-11 C

Procedure	Description
lib$get_foreign	Get DCL command line
lib$free_vm	Virtual memory deallocation
lib$get_vm	Virtual memory allocation
lib$signal	Condition signaling

The VAX-11 C mathematical functions are performed by the VAX/VMS run-time procedures listed below:

mth$dacos_r7	mth$dasin_r7	mth$datan_r7
mth$datan2	mth$dcos_r7	mth$dcosh
mth$dexp_r6	mth$dsqrt_r5	mth$dlog_r8
mth$dlog10_r8	mth$dsin_r7	mth$dsinh
mth$dsqrt_r5	mth$dtan_r7	mth$dtanh

VAX-11 C also calls run-time library modules that perform data conversion. These modules are listed below:

ots$cvt_t_d
ots$cvt_ti_l
ots$cvt_to_l
ots$cvt_tz_l
ots$$cvt_d_t_r8
ots$powdd

The following formatting routines are called by VAX-11 C:

for$cvt_d_tg
for$cvt_d_te
for$cvt_d_tf

Appendix G

ASCII Character Set

Table G-1 shows the ASCII character set and the values returned by the character classification functions.

Along the top of the table are the names of the functions. (Each function name begins with "is"; the "is" was dropped to avoid unnecessary redundancy.) Along the left side of the table are the ASCII characters and their octal values. A check mark (✓) in the column under the name of a function means that that function returns a true (nonzero) value when the corresponding ASCII character is the argument to the function. (A blank means that the function returns a false value.)

Note that these functions are implemented as preprocessor macros. Chapter 6 describes each character classification function in detail.

Table G-1: ASCII Character Set (with character-classification return values)

ASCII Code		is-Function											
		alnum	alpha	ascii	cntrl	digit	graph	lower	print	punct	space	upper	xdigit
NUL	00			✓	✓								
SOH	01			✓	✓								
STX	02			✓	✓								
ETX	03			✓	✓								
EOT	04			✓	✓								
ENQ	05			✓	✓								
ACK	06			✓	✓								
BEL	07			✓	✓								
BS	10			✓	✓								
HT	11			✓	✓						✓		
LF	12			✓	✓						✓		
VT	13			✓	✓						✓		
FF	14			✓	✓						✓		
CR	15			✓	✓						✓		
SO	16			✓	✓								
SI	17			✓	✓								
DLE	20			✓	✓								
DC1	21			✓	✓								
DC2	22			✓	✓								
DC3	23			✓	✓								

Table G-1: (Cont.) ASCII Character Set (with character-classification return values)

ASCII Code		is-Function											
		alnum	alpha	ascii	cntrl	digit	graph	lower	print	punct	space	upper	xdigit
DC4	24			✓	✓								
NAK	25			✓	✓								
SYN	26			✓	✓								
ETB	27			✓	✓								
CAN	30			✓	✓								
EM	31			✓	✓								
SUB	32			✓	✓								
ESC	33			✓	✓								
FS	34			✓	✓								
GS	35			✓	✓								
RS	36			✓	✓								
US	37			✓	✓								
SP	40			✓					✓		✓		
!	41			✓			✓		✓	✓			
"	42			✓			✓		✓	✓			
#	43			✓			✓		✓	✓			
$	44			✓			✓		✓	✓			
%	45			✓			✓		✓	✓			
&	46			✓			✓		✓	✓			
'	47			✓			✓		✓	✓			
(	50			✓			✓		✓	✓			
)	51			✓			✓		✓	✓			
*	52			✓			✓		✓	✓			
+	53			✓			✓		✓	✓			
,	54			✓			✓		✓	✓			
-	55			✓			✓		✓	✓			
.	56			✓			✓		✓	✓			
/	57			✓			✓		✓	✓			
0	60	✓		✓		✓	✓		✓				✓
1	61	✓		✓		✓	✓		✓				✓
2	62	✓		✓		✓	✓		✓				✓
3	63	✓		✓		✓	✓		✓				✓
4	64	✓		✓		✓	✓		✓				✓
5	65	✓		✓		✓	✓		✓				✓
6	66	✓		✓		✓	✓		✓				✓
7	67	✓		✓		✓	✓		✓				✓
8	70	✓		✓		✓	✓		✓				✓
9	71	✓		✓		✓	✓		✓				✓
:	72			✓			✓		✓	✓			
;	73			✓			✓		✓	✓			
<	74			✓			✓		✓	✓			
=	75			✓			✓		✓	✓			
>	76			✓			✓		✓	✓			
?	77			✓			✓		✓	✓			
@	100			✓			✓		✓	✓			

Table G–1: (Cont.) ASCII Character Set (with character-classification return values)

ASCII Code		is-Function											
		alnum	alpha	ascii	cntrl	digit	graph	lower	print	punct	space	upper	xdigit
A	101	✓	✓	✓			✓		✓			✓	✓
B	102	✓	✓	✓			✓		✓			✓	✓
C	103	✓	✓	✓			✓		✓			✓	✓
D	104	✓	✓	✓			✓		✓			✓	✓
E	105	✓	✓	✓			✓		✓			✓	✓
F	106	✓	✓	✓			✓		✓			✓	✓
G	107	✓	✓	✓			✓		✓			✓	
H	110	✓	✓	✓			✓		✓			✓	
I	111	✓	✓	✓			✓		✓			✓	
J	112	✓	✓	✓			✓		✓			✓	
K	113	✓	✓	✓			✓		✓			✓	
L	114	✓	✓	✓			✓		✓			✓	
M	115	✓	✓	✓			✓		✓			✓	
N	116	✓	✓	✓			✓		✓			✓	
O	117	✓	✓	✓			✓		✓			✓	
P	120	✓	✓	✓			✓		✓			✓	
Q	121	✓	✓	✓			✓		✓			✓	
R	122	✓	✓	✓			✓		✓			✓	
S	123	✓	✓	✓			✓		✓			✓	
T	124	✓	✓	✓			✓		✓			✓	
U	125	✓	✓	✓			✓		✓			✓	
V	126	✓	✓	✓			✓		✓			✓	
W	127	✓	✓	✓			✓		✓			✓	
X	130	✓	✓	✓			✓		✓			✓	
Y	131	✓	✓	✓			✓		✓			✓	
Z	132	✓	✓	✓			✓		✓			✓	
[	133			✓			✓		✓	✓			
\	134			✓			✓		✓	✓			
]	135			✓			✓		✓	✓			
^	136			✓			✓		✓	✓			
_	137			✓			✓		✓	✓			
`	140			✓			✓		✓	✓			
a	141	✓	✓	✓			✓	✓	✓				✓
b	142	✓	✓	✓			✓	✓	✓				✓
c	143	✓	✓	✓			✓	✓	✓				✓
d	144	✓	✓	✓			✓	✓	✓				✓
e	145	✓	✓	✓			✓	✓	✓				✓
f	146	✓	✓	✓			✓	✓	✓				✓
g	147	✓	✓	✓			✓	✓	✓				
h	150	✓	✓	✓			✓	✓	✓				
i	151	✓	✓	✓			✓	✓	✓				
j	152	✓	✓	✓			✓	✓	✓				
k	153	✓	✓	✓			✓	✓	✓				
l	154	✓	✓	✓			✓	✓	✓				
m	155	✓	✓	✓			✓	✓	✓				

Table G-1: (Cont.) ASCII Character Set (with character-classification return values)

ASCII Code		is-Function											
		alnum	alpha	ascii	cntrl	digit	graph	lower	print	punct	space	upper	xdigit
n	156	✓	✓	✓			✓	✓	✓				
o	157	✓	✓	✓			✓	✓	✓				
p	160	✓	✓	✓			✓	✓	✓				
q	161	✓	✓	✓			✓	✓	✓				
r	162	✓	✓	✓			✓	✓	✓				
s	163	✓	✓	✓			✓	✓	✓				
t	164	✓	✓	✓			✓	✓	✓				
u	165	✓	✓	✓			✓	✓	✓				
v	166	✓	✓	✓			✓	✓	✓				
w	167	✓	✓	✓			✓	✓	✓				
x	170	✓	✓	✓			✓	✓	✓				
y	171	✓	✓	✓			✓	✓	✓				
z	172	✓	✓	✓			✓	✓	✓				
{	173			✓			✓		✓	✓			
|	174			✓			✓		✓	✓			
}	175			✓			✓		✓	✓			
~	176			✓			✓		✓	✓			
DEL	177			✓	✓								

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

PROGRAMMING IN VAX–11 C **READER'S COMMENTS**
AA-L370A-TE

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.)

What features are most useful?

Does the publication satisfy your needs?

What errors have you found?

Additional comments

Name

Title

Company Dept.

Address

City State Zip

(staple here)

— — — — Do Not Tear - Fold Here — — — — — — — — — — — — —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

(staple here)

Cut Along Dotted Line

AA-L370A-TE