April 1980

This document describes the FORTRAN language elements supported by VAX-11 FORTRAN. It is intended to be used as a reference manual in preparing FORTRAN source programs. It is not a tutorial document, nor does it present information on the FORTRAN user's interface to the VAX/VMS operating system.

VAX-11 FORTRAN

Language Reference Manual

Order No. AA-D034B-TE

SUPERSESSION/UPDATE INFORMATION: This document supersedes AA-D034A-TE.

OPERATING SYSTEM AND VERSION: VAX/VMS V2.0

SOFTWARE VERSION: VAX-11 FORTRAN IV-PLUS V2.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1980
Revised, April 1980

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

9/80-15

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

The primary objective of this document is to provide the FORTRAN programmer with a complete description of the elements of DIGITAL'S VAX-11 FORTRAN language; it is not designed to serve as a tutorial document.

The reader will find detailed instructions on the use of VAX-11 FORTRAN in the companion manual to this document: the VAX-11 FORTRAN User's Guide.

INTENDED AUDIENCE

This manual is intended for readers who have a basic understanding of the FORTRAN language. It is not necessary for the reader to have a detailed understanding of the VAX/VMS operating system, but some familiarity with VAX/VMS is helpful. For information concerning VAX/VMS, refer to the documents listed below under "ASSOCIATED DOCUMENTS."

STRUCTURE OF THIS DOCUMENT

This manual contains nine chapters and three appendixes. These are summarized below:

- Chapter 1, "Introduction to VAX-11 FORTRAN," contains general information concerning FORTRAN, and introduces basic facts needed prior to writing FORTRAN programs.
- Chapter 2, "FORTRAN Statement Components," describes the components of FORTRAN statements, such as symbols, constants, variables, etc.
- Chapter 3, "Assignment Statements," describes assignment statements, which define values used in the program.
- Chapter 4, "Control Statements," deals with control statements, which transfer control from one point in the program to another.
- Chapter 5, "Specification Statements," describes specification statements, which define characteristics of symbols used in the program, such as data type, array dimensions, etc.
- Chapter 6, "Subprograms," discusses subprograms: both user-written and those supplied with VAX-11 FORTRAN.

- Chapter 7, "Input/Output Statements," covers FORTRAN input/output.
- Chapter 8, "Format Statements," describes the FORMAT statements used in conjunction with formatted I/O statements.
- Chapter 9, "Auxiliary Input/Output Statements," contains information on auxiliary I/O statements, such as OPEN, CLOSE, and UNLOCK.
- Appendix A describes some additional statements and language features that provide compatible support for programs written in older versions of FORTRAN; the statements include ENCODE, DECODE, DEFINE FILE, FIND, and alternative PARAMETER and EXTERNAL statements.
- Appendix B summarizes the character sets supported by VAX-11 FORTRAN.
- Appendix C summarizes the language elements of VAX-11 FORTRAN.

ASSOCIATED DOCUMENTS

The following documents are of interest to VAX-11 FORTRAN programmers:

- VAX/VMS Primer
- VAX-11 FORTRAN User's Guide
- VAX/VMS Command Language User's Guide

For a complete list of all VAX-11 documents, including brief descriptions of each, see the VAX-11 Information Directory.

CONVENTIONS USED IN THIS DOCUMENT

The following syntactic conventions are used in this manual:

- Upper-case words and letters used in examples indicate that you should type the word or letter as shown.
- Lower-case words and letters used in format examples indicate that you are to substitute a word or value of your choice.
- Brackets ([]) indicate optional elements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times.
- "Real" (lower case) is used to refer to the REAL*4 (REAL), REAL*8, and REAL*16 data types as a group; likewise, "complex" (lower case) is used to refer to the COMPLEX*8 (COMPLEX) and COMPLEX*16 data types as a group; "logical" (lower case) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group; and "integer" (lower case) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.

In addition, the following notations are used to denote special nonprinting characters:

Tab character (TAB)

Space character △

or

(SP) (in terminal dialog examples)

CHAPTER 1

INTRODUCTION TO VAX-11 FORTRAN

1.1 LANGUAGE OVERVIEW

VAX-11 FORTRAN[1] is based on American National Standard FORTRAN-77 (ANSI X3.9-1978). It includes support for programs that conform to the previous standard (ANSI X3.9-1966), as well as the following enhancements to FORTRAN-77:

- You can use any arithmetic expression as an array subscript. If the expression is not of integer type, it is converted to integer type.
- Mixed-mode expressions can contain elements of any data type, including complex.
- The following data types have been added:

```
LOGICAL*1
LOGICAL*2
INTEGER*2
REAL*16
COMPLEX*16
```

- The following device-oriented input/output (I/O) statements have been added:

```
ACCEPT
TYPE
```

- The following I/O statements provide keyed access and operations on indexed files:

```
READ (unit,kspec)       Unformatted and formatted indexed
READ (unit,fmt,kspec)   I/O
WRITE (unit)
WRITE (unit,fmt)

DELETE (unit)           ISAM file manipulation
REWRITE (unit)
REWRITE (unit,fmt)
UNLOCK (unit)
```

---

1. VAX-11 FORTRAN is referred to simply as FORTRAN throughout the rest of this manual.

- The following I/O statements provide compatible support for older FORTRAN programs:

DEFINE FILE	File control and attribute specification
ENCODE DECODE	Formatted data conversion in memory

- The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.
- You can include an explanatory comment on the same line with any FORTRAN statement. These comments begin with exclamation points (!).
- You can include debugging statements in a program by placing the letter D in column 1. These statements are compiled only when the associated compiler command qualifier is specified; otherwise, they are treated as comments.
- A compiler command qualifier selects either the VAX-11 D_floating or G_floating hardware data type as the implementation of REAL*8.
- Data initialization is permitted in type declaration statements.
- You can use octal and hexadecimal constants in place of any numeric constants.
- Symbolic names can be up to 31 characters long and consist of letters, digits, dollar signs ($), and underline characters (_).

VAX-11 FORTRAN is also a compatible superset of PDP-11 FORTRAN IV-PLUS. This means you can compile existing PDP-11 FORTRAN IV-PLUS source programs as well as new programs that incorporate features available in VAX-11 FORTRAN.

1.2 ELEMENTS OF FORTRAN PROGRAMS

FORTRAN programs consist of FORTRAN statements and optional comments. The statements are organized into program units. A program unit is a sequence of statements that define a computing procedure and is terminated by an END statement. Also, a program unit can be either a main program or a subprogram. An executable program consists of one main program and, optionally, one or more subprograms.

1.2.1 Statements

Statements are grouped into two general classes: executable and nonexecutable. Executable statements describe the action of the program. Nonexecutable statements describe data arrangement and characteristics, and provide editing and data-conversion information.

Statements are divided into physical sections called lines. A line is a string of up to 80 characters. If a statement is too long to fit on one line, you can continue it on one or more additional lines called

continuation lines. A continuation line is identified by a continuation character in the sixth column of that line. (For further information on continuation characters, see Section 1.3.4.)

You can identify a statement with a statement label so that other statements can refer to it, either for the information it contains or to transfer control to it. A statement label takes the form of an integer number in the first five columns of a statement's initial line. Any statement can have a label; however, only executable and FORMAT statements can be referred to.

1.2.2 **Comments**

Comments do not affect program processing in any way. They are merely a documentation aid to the programmer. You can use them freely to describe the actions of the program, to identify program sections and processes, and to provide greater ease in reading the source program listing. The letter C or an asterisk (*) in the first column of a source line identifies that line as a comment; a line containing only spaces is also a comment line. In addition, if you place an exclamation point (!) in column 1 or in the statement portion of a source line, the rest of that line is treated as a comment.

1.2.3 **FORTRAN Character Set**

The FORTRAN character set consists of:

1. All upper-case and lower-case letters (A through Z, a through z)
2. The numerals 0 through 9
3. The special characters listed below

Character	Name
△ or TAB	Space or tab
=	Equal sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Period
'	Apostrophe
"	Quotation mark

Character	Name
$	Dollar sign
_	Underline
!	Exclamation point
:	Colon
<	Left angle bracket
>	Right angle bracket
%	Percent sign
&	Ampersand

Other printable ASCII characters can appear in a FORTRAN statement only as part of a character or Hollerith constant (see Appendix B for a list of printable characters). Any printable character can appear in a comment.

Except in character and Hollerith constants, the compiler makes no distinction between upper-case and lower-case letters.

1.3 FORMATTING A FORTRAN LINE

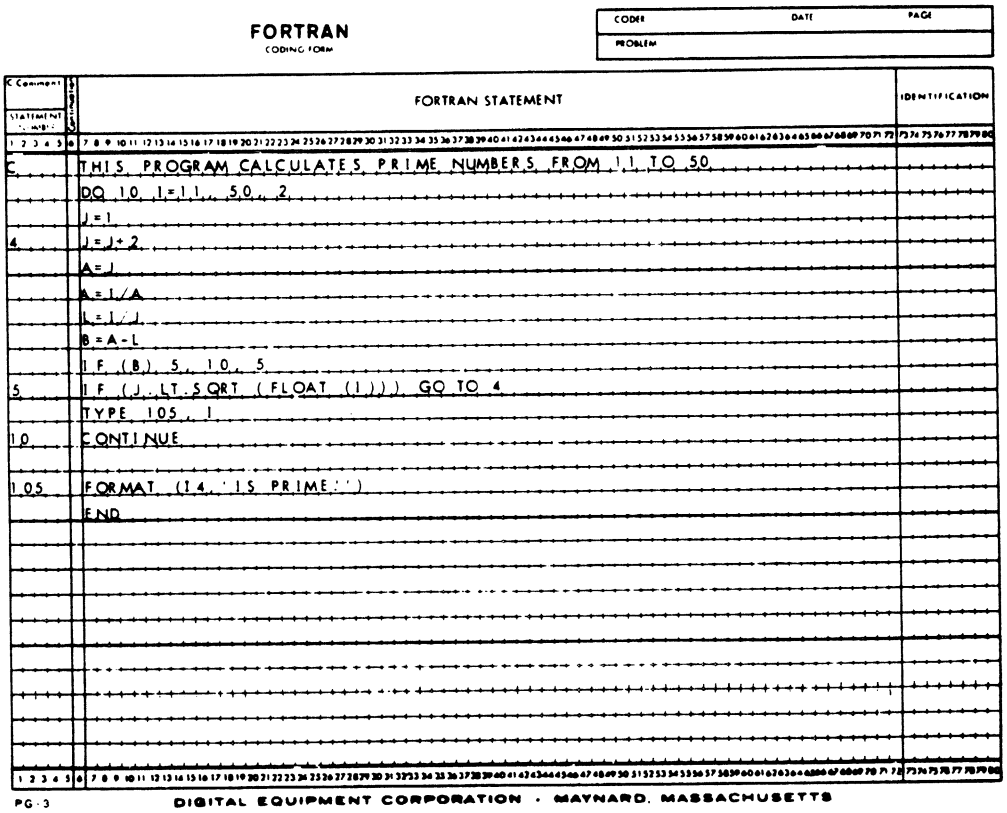
Each FORTRAN line has the following four fields:

- Statement label field
- Continuation indicator field
- Statement field
- Sequence number field

There are two ways to format a FORTRAN line: 1) on a character-per-column basis; or 2) by using the tab character. You can use character-per-column formatting when punching cards, using a coding form, or entering lines from a terminal with a text editor. You can use tab-character formatting when you are entering lines at a terminal with a text editor.

1.3.1 Character-per-Column Formatting

As shown in Figure 1-1, a FORTRAN line is divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character. Sections 1.3.3 through 1.3.6 describe the use of each field.



FORTRAN
CODING FORM

CODER | DATE | PAGE
PROBLEM

C Comment / STATEMENT NUMBER	Continuation	FORTRAN STATEMENT	IDENTIFICATION
1 2 3 4 5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80
C		THIS PROGRAM CALCULATES PRIME NUMBERS FROM 1 TO 50	
		DO 10 I=1,1,50,2	
		J=1	
4		J=J+2	
		A=J	
		A=1/A	
		L=I/J	
		B=A-L	
		IF (B) 5,10,5	
5		IF (J.LT.SQRT (FLOAT (I))) GO TO 4	
		TYPE 105, I	
10		CONTINUE	
105		FORMAT (I4,' IS PRIME')	
		END	

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

PG-3 DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS

Figure 1-1 FORTRAN Coding Form

To enter an item in a field, enter it in the column(s) in the coding form, as listed below:

Field	Column(s)
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72
Sequence number	73 through 80

1.3.2 Tab-Character Formatting

You can use tab-character formatting to specify the statement label field, the continuation indicator field, and the statement field. However, you cannot specify a sequence number field with tab-character formatting. Figure 1-2 illustrates FORTRAN lines with tab-character formatting and the equivalent lines with character-per-column formatting.

Format Using TAB Character

C (TAB) FIRST VALUE

10 (TAB) I = J + 5*K +

(TAB) 1 L*M

(TAB) IVAL = I+2

Character-per-Column Format

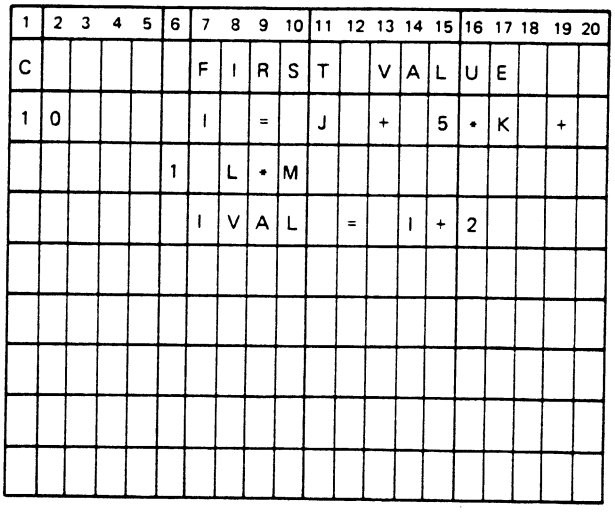


Figure 1-2 Line Formatting Example

The statement label field consists of the characters that you type before the first tab character. The statement label field cannot have more than five characters.

After you type the first tab character, you can type either the continuation indicator field or the statement field.

To enter the continuation indicator field, type any digit after the first tab. If you enter the continuation indicator field, the statement field consists of all the characters after the digit to the end of the line.

To enter the statement field without a continuation indicator field, type the statement immediately after the first tab. Note that no FORTRAN statement starts with a digit.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you type the TAB key. However, this action is not related to the FORTRAN compiler's interpretation of the tab character described above.

You can use the space character to improve the legibility of a FORTRAN statement. The compiler ignores all spaces in a statement field except those within a character or Hollerith constant. For example, GO TO and GOTO are equivalent. The compiler treats the tab character in a statement field the same as a space. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (located at columns 9, 17, 25, 33, etc.).

1.3.3 Statement Label Field

Any statement can have a label. A statement label or statement number consists of one to five decimal digits in the statement label field of a statement's initial line. Spaces and leading zeros are ignored. An all-zero statement label is invalid.

The only statements that can be referred to by other statements are labelled FORMAT and executable statements (see Section 1.4). FORMAT statements are referred to only in the format specifier of an I/O statement, or in an ASSIGN statement. No two statements within a program unit can have the same label.

You can use two special indicators -- the comment indicator and the debugging statement indicator -- in the first column of the label field. These indicators are described below.

The statement label field of a continuation line must be blank.

1.3.3.1 **Comment Indicator** - You can use the letter C, an asterisk (*), or an exclamation point (!) in column 1 to indicate that the line is a comment. The compiler prints that line in the source program listing, and then ignores the line. An all-blank line is also considered to be a comment.

1.3.3.2 **Debugging Statement Indicator** - You can use the letter D in column 1 to designate debugging statements. The initial line of the debugging statement can contain a statement label in the remaining columns of the label field. If a debugging statement is continued onto more than one line, every continuation line must contain a D (in column 1) and a continuation indicator.

The compiler treats the debugging statement either as source text to be compiled or as a comment, depending on the setting of the /D_LINES compiler command qualifier. If you specify /D_LINES, debugging statements are compiled as a part of the source program; if you do not specify /D_LINES, debugging statements are treated as comments.

1.3.4 Continuation Field

A continuation indicator is any character, except zero or space, in column 6 of a FORTRAN line or any digit, except zero, after the first tab; it divides a statement into distinct lines at any point. The compiler considers the characters after the continuation character to be the characters following the last character of the previous line, as if there were no break at that point. If a continuation indicator is zero or space, the compiler considers the line to be an initial line of a FORTRAN statement.

Comment lines cannot be continued. They can occur between a statement's initial line and its continuation line(s), or between successive continuation lines.

1.3.5 Statement Field

The text of a FORTRAN statement is placed in the statement field. Because the compiler ignores the tab character and spaces (except in

character and Hollerith constants), you can space the text in any way desired for maximum legibility.

NOTE

If a line extends beyond character position 72, the text following position 72 is ignored and no warning message is printed.

1.3.6 Sequence Number Field

A sequence number or other identifying information can appear in columns 73 through 80 of any line in a FORTRAN program. The compiler ignores the characters in this field.

1.4 PROGRAM UNIT STRUCTURE

Figure 1-3 shows the order of statements in a FORTRAN program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, DATA statements can be interspersed with executable statements. On the other hand, horizontal lines indicate statement types that cannot be interspersed. For example, type declaration statements cannot be interspersed with executable statements.

Executable statements include: assignment statements, ASSIGN, GOTO, computed GOTO, assigned GOTO, arithmetic IF, logical IF, block IF, ELSE, ENDIF, CONTINUE, STOP, PAUSE, DO, ENDDO, READ, WRITE, PRINT, TYPE, ACCEPT, FIND, DELETE, REWRITE, BACKSPACE, ENDFILE, REWIND, UNLOCK, OPEN, CLOSE, INQUIRE, CALL, RETURN, and END.

Specification statements include: DIMENSION, COMMON, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE, and type declarations.

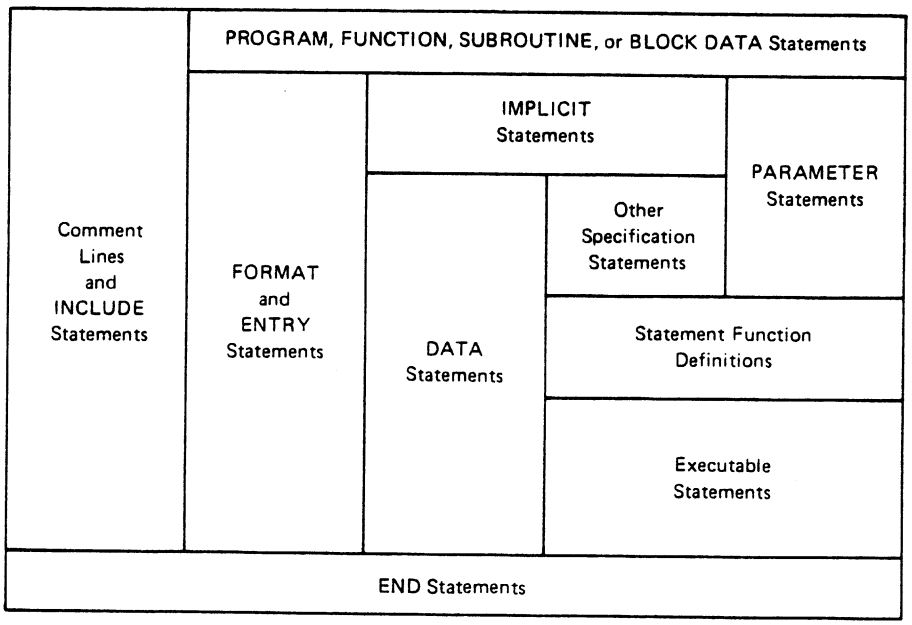


Figure 1-3 Required Order of Statements and Lines

INCLUDE

1.5 INCLUDE STATEMENT

The INCLUDE statement specifies that the contents of a designated file are to be incorporated in the FORTRAN compilation directly following the INCLUDE statement. The INCLUDE statement is described in this chapter rather than with the other FORTRAN statements because it has no effect on program execution, except to direct the compiler to read FORTRAN statements from a file.

The INCLUDE statement has the form

```
INCLUDE 'file specification[/[NO]LIST]'
```

where:

file specification
is a character constant that specifies the file to be included in the compilation. This file specification must be acceptable to the operating system. (See the VAX-11 FORTRAN User's Guide for the form of a file specification.)

The /LIST qualifier indicates that the statements in the specified file are to be listed in the compilation source listing. An asterisk (*) precedes each statement listed. The /NOLIST qualifier indicates that the included statements are not to be listed in the compilation source listing. The default is /LIST; that is, the compiler assumes /LIST if you do not specify a qualifier.

When the compiler encounters an INCLUDE statement, it stops reading statements from the current file and reads the statements in the included file. When it reaches the end of the included file, the compiler resumes compilation with the next statement after the INCLUDE statement.

An INCLUDE statement can be contained in an included file.

An included file cannot begin with a continuation line. Each FORTRAN statement must be completely contained within a single file.

The INCLUDE statement can appear anywhere within a program unit, as shown in Figure 1-3. Any FORTRAN statement can appear in an included file. However, the included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions described in Section 1.4.

In the following example, the file COMMON.FOR defines the size of the blank COMMON block and the size of the arrays X, Y, and Z.

Main Program File

```
      INCLUDE 'COMMON.FOR'
      DIMENSION Z(M)
      CALL CUBE
      DO 5, I=1,M
5     Z(I) = X(I)+SQRT(Y(I))
           .
           .
           .
      END

      SUBROUTINE CUBE
      INCLUDE 'COMMON.FOR'
      DO 10, I=1,M
10    X(I) = Y(I)**3
      RETURN
      END
```

File COMMON.FOR

```
PARAMETER (M = 100)
COMMON X(M),Y(M)
```

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

The basic components of FORTRAN statements are:

- Constants -- fixed, self-describing values.
- Variables -- symbolic names that represent stored values.
- Arrays -- groups of values that are stored contiguously and can be referred to individually or collectively. Individual values are called array elements.
- Expressions -- single constants, variables, array elements, or function references; or, combinations of these components plus certain other elements, called operators, that specify computations to be performed on the values of these components to obtain a single result.
- Function references -- names of functions, optionally followed by lists of arguments. A function reference performs the computation indicated by the function definition. The resulting value is used in place of the function reference.

Variables, arrays, and functions have symbolic names. A symbolic name is a string of characters that identify entities in the program.

Constants, variables, arrays, expressions, and functions can have the following general data types:

- Logical
- Integer
- Real (includes REAL*4, REAL*8, and REAL*16)
- Complex (includes COMPLEX*8 and COMPLEX*16)
- Character

Sections 2.1 and 2.2 discuss symbolic names and data types; the remaining sections detail the basic components of FORTRAN, with the exception of function references, which are described in Chapter 6.

2.1 SYMBOLIC NAMES

Symbolic names identify entities within a FORTRAN program unit. These entities are listed in Table 2-1.

A symbolic name is a string of letters, digits, and the dollar sign ($) and underline (_) special characters. The first character in a symbolic name must be a letter. The symbolic name can contain a maximum of 31 characters.

Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(begins with a numeral)
K9	B.4	(contains a special character
X		other than _ or $)
FIND_IT	$FREQ	(begins with a $)

By convention, symbolic names containing a dollar sign ($) are reserved for use in DIGITAL-supplied software components. To avoid name conflicts, you should not define any symbolic names in your program that contain a dollar sign.

Symbolic names must be unique within a program unit. That is, you cannot use the same symbolic name to identify two or more entities in the same program unit. Furthermore, in an executable program consisting of two or more program units, the symbolic names of the following entities must be unique within the entire program:

- Processor-defined functions
- Function subprograms
- Subroutine subprograms
- Common blocks
- Main programs
- Block data subprograms
- Function entries
- Subroutine entries

That is, if your program contains a function named BTU, you cannot use BTU as the symbolic name of any other subprogram, entry, or common block in the program, even if the name appears in a different program unit.

Each entity with "yes" under "Typed" in Table 2-1 has a data type. Sections 2.4.1 and 2.4.2 discuss how to specify the data type of a name.

Within a subprogram, you can also use symbolic names as dummy arguments. A dummy argument can represent a variable, array, array element, constant, expression, or subprogram.

Table 2-1
Entities Identified by Symbolic Names

Entity	Typed
Variables	yes
Arrays	yes
Statement functions	yes
Intrinsic functions	yes
Function subprograms	yes
Subroutine subprograms	no
Common blocks	no
Main programs	no
Block data subprograms	no
Function entries	yes
Subroutine entries	no
Parameter constants	yes

2.2 DATA TYPES

Each basic component represents data of one of several types. The data type of a component can be inherent in its construction, implied by convention, or explicitly declared. The data types available in FORTRAN, and their definitions, are:

- Integer -- a whole number
- REAL*4 (REAL) -- a decimal number, that is, a whole number, a decimal fraction, or a combination of the two
- REAL*8 (DOUBLE PRECISION) -- similar to REAL*4, but with more than twice the degree of accuracy in its representation
- REAL*16 -- similar to REAL*4; has an extended range, and more than four times the accuracy in its representation
- COMPLEX*8 (COMPLEX) -- a pair of REAL*4 values that represent a complex number; the first value represents the real part of that number, and the second represents the imaginary part
- COMPLEX*16 (DOUBLE COMPLEX) -- similar to complex; its real and imaginary parts are REAL*8
- Logical -- the logical value, true or false
- Character -- a sequence of characters

2.2.1 Storage Requirements

An important attribute of each data type is the amount of memory required to represent a value of that type. Variations on the basic types affect either the accuracy of the represented value or the allowed range of values.

ANSI FORTRAN defines a "numeric storage unit" as the amount of storage needed to represent a real, integer, or logical value. In VAX-11 FORTRAN, a numeric storage unit corresponds to four bytes of memory. REAL*8 and COMPLEX*8 values occupy two of these numeric storage units, whereas REAL*16 and COMPLEX*16 values occupy four.

ANSI FORTRAN defines a "character storage unit" as the amount of storage needed to represent one character value. In VAX-11 FORTRAN, a character storage unit corresponds to one byte of memory.

VAX-11 FORTRAN provides additional data types for optimum selection of performance and memory requirements. Table 2-2 lists the data types available, the names associated with each data type, and the amount of storage required (in bytes). The form *n appended to a data type name is called a data type length specifier.

Table 2-2
Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1 [2]
LOGICAL	2 or 4 [1]
LOGICAL*1	1 [2]
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4 [1]
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
REAL*16	16
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16
DOUBLE COMPLEX	16
CHARACTER*len CHARACTER*(*)	len [3]

1. Either two or four bytes are allocated depending on the compiler command qualifier specified. The default allocation is four bytes.

2. The 1-byte storage area can contain the logical values true or false, a single character, or integers in the range -128 to +127.

3. The value of len is the number of characters specified, Which can be in the range 1 to 32767. Passed length format -- *(*) -- applies to dummy arguments or character functions, and indicates that the length of the actual argument or function is used (see Chapter 6).

2.2.2 VAX Implementations of REAL*8

There are two implementations of the REAL*8 (and COMPLEX*16) data type on VAX-11: D_floating and G_floating. The G_floating implementation offers a greater range than the D_floating implementation but a smaller number of significant digits. You select the G_floating implementation by compiling the program under the /G_FLOATING compiler command qualifier; the default implementation of REAL*8 is D_floating.

Programs compiled with the /G_FLOATING qualifier can only be executed on VAX-11 machines that are equipped with the extended floating-point option.

See Section 2.3.3 for more detailed information on the two implementations of the REAL*8 data type.

2.3 CONSTANTS

A constant represents a fixed value and can be a numeric value, a logical value, or a character string.

Octal, hexadecimal, and Hollerith constants have no data type. They assume a data type that conforms to the context in which they appear (see Sections 2.3.5 and 2.3.8).

2.3.1 Integer Constants

An integer constant is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

An integer constant has the form

```
snn
```

where:

s
is an optional sign.

nn
is a string of numeric characters.

Leading zeros, if any, are ignored.

A minus sign must appear before a negative integer constant, whereas a plus sign is optional before a positive constant (an unsigned constant is assumed to be positive).

Except for a leading algebraic sign, an integer constant cannot contain any character other than the numerals 0 through 9.

The value of an integer constant must be within the range -2147483648 to +2147483647.

Examples of valid and invalid integer constants are:

Valid	Invalid	
0	99999999999	(too large)
-127	3.14	(decimal point and
+32123	32,767	comma not allowed)

If the value of the constant is within the range -32768 to +32767, it represents a 2-byte signed quantity and is treated as INTEGER*2 data type. If the value is outside that range, it represents a 4-byte signed quantity and is treated as INTEGER*4 data type.

Integer constants can also be specified in octal form. See Section A.5.

2.3.2 REAL*4 (REAL) Constants

A REAL*4 constant can be any one of the following:

- A basic real constant
- A basic real constant followed by a decimal exponent
- An integer constant followed by a decimal exponent

Integer constants are defined in the preceding subsection. A basic real constant is a string of decimal digits having one of the following forms:

```
  s.nn
snn.nn
snn.
```

where:

s
is an optional sign.

nn
is a string of numeric characters (decimal digits).

The decimal point can appear anywhere in the string. The number of digits is not limited, but typically only the leftmost seven digits are significant. Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting the leftmost seven digits. Thus, in the constant 0.00001234567, all the nonzero digits, and none of the zeroes, are significant.

A decimal exponent has the form

```
Esnn
```

where:

s
is an optional sign.

nn
is an integer constant.

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied (for example, 1.0E6 represents the value 1.0 * 10 ** 6).

A REAL*4 constant occupies four bytes of VAX-11 storage and is interpreted as a real number with a degree of precision that is typically seven decimal digits.

A minus sign must appear before a negative REAL*4 constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the letter E and a negative exponent, whereas a plus sign is optional between the letter E and a positive exponent.

Except for algebraic signs, a decimal point, and the letter E (if used), a REAL*4 constant cannot contain any character other than the numerals 0 through 9.

If the letter E appears in a REAL*4 constant, an integer constant exponent field must follow. The exponent field cannot be omitted; it can, however, be zero.

The magnitude of a nonzero REAL*4 constant cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Examples of valid and invalid REAL*4 constants follow:

Valid	Invalid	
3.14159	1,234,567	(commas not allowed)
621712.	325E-45	(too small)
-.00127	-47.E47	(too large)
+5.0E3	100	(decimal point missing)
2E-3	$25.00	(special character not allowed)

2.3.3 REAL*8 (DOUBLE PRECISION) Constants

A REAL*8 constant is a basic real constant or an integer constant followed by a decimal exponent of the form

Dsnn

where:

s
is an optional sign.

nn
is an integer constant.

There are two implementations of the REAL*8 constant: D_floating and G_floating. Both implementations have the same syntax and storage requirements, but each has a different number of significant digits and a different exponent range. The G_floating implementation is invoked with the /G_FLOATING compiler command qualifier.

A REAL*8 constant occupies 8 bytes of VAX-11 storage and is interpreted as a real number with a degree of precision that is typically 16 decimal digits for D_floating and 15 for G_floating.

A minus sign must appear before a negative REAL*8 constant; a plus sign is optional before a positive constant. Similarly, a minus sign

must appear between the letter D and a negative exponent, whereas a plus sign is optional between the letter D and a positive exponent.

The exponent field following the letter D cannot be omitted; it can, however, be zero.

The magnitude of a nonzero REAL*8 constant cannot be less than approximately 0.29D-38 or greater than approximately 1.7D38 for the D_floating implementation, or less than approximately 0.56D-308 or greater than approximately 0.9D308 for the G_floating implementation.

Examples of valid and invalid D_floating and G_floating REAL*8 constants follow:

D_floating

Valid	Invalid	
1234567890D+5	1234567890D45	(too large)
+2.71828182846182D00	1234567890.0D-89	(too small)
-72.5D-15	+2.7182812846182	(no Dsnn present; this is a valid single precision constant)
1D0		

G_floating

Valid	Invalid	
123456789.D0	123456789.D400	(too large)
+2.34567890123D-5	123456789.D-400	(too small)
-1D+300		

NOTE

Programs that use the G_floating implementation of the REAL*8 data type can only be executed on VAX-11 machines that are equipped with the extended floating-point option.

2.3.4 REAL*16 Constants

A REAL*16 constant is a basic real constant or an integer constant followed by a decimal exponent of the form

Qsnn

where:

s
is an optional sign.

nn
is an integer constant.

The number of digits that precede the exponent is not limited, but typically only the leftmost 33 digits are significant.

A REAL*16 constant occupies 16 bytes of VAX-11 storage and is interpreted as a real number with a degree of precision that is typically 33 decimal digits.

A minus sign must appear before a negative REAL*16 constant; a plus sign is optional before a positive constant. Similarly, a minus sign is required between the letter Q and a negative exponent, whereas a plus sign is optional between the letter Q and a positive exponent.

The exponent field following the letter Q cannot be omitted; it can, however, be zero.

The magnitude of a nonzero REAL*16 constant cannot be less than approximately 0.84Q-4932 or greater than approximately 0.59Q4932.

Examples of valid and invalid REAL*16 constants follow:

Valid	Invalid
1234567890Q4000	1.Q5000 (too large)
-1.23Q-400	1.Q-5000 (too small)
+2.72Q0	

NOTE

Programs that use the REAL*16 data type can only be executed on VAX-11 machines that are equipped with the extended floating-point option.

2.3.5 COMPLEX*8 (COMPLEX) Constants

A COMPLEX*8 constant is a pair of integer or REAL*4 constants that represent a complex number. The two constants are separated by a comma and enclosed in parentheses. The first constant represents the real part of that number and the second constant represents the imaginary part.

A COMPLEX*8 constant has the form

(c,c)

where:

c
is an integer or REAL*4 constant.

The parentheses and comma are part of the constant and are required. See Section 2.3.2 for the rules for forming REAL*4 constants.

A COMPLEX*8 constant occupies eight bytes of VAX-11 storage and is interpreted as a complex number.

Examples of valid and invalid COMPLEX constants are:

Valid	Invalid
(1.7039,-1.70391)	(1.23,) (second REAL constant is missing)
(+12739E3,0.)	(1.0,1.0Q0) (REAL*16 constants are not allowed)
(1,2)	

2.3.6 COMPLEX*16 Constants

A COMPLEX*16 constant is a pair of constants that represents a complex number. One of this pair must be REAL*8, and the other must be integer, REAL*4, or REAL*8. The two constants are separated by a comma and enclosed by parentheses; the first constant represents the real part of the complex number, the second the imaginary part. There are two implementations of COMPLEX*16, corresponding to the D_floating and G_floating implementations of REAL*8.

A COMPLEX*16 constant has the form

```
(c,c)
```

where:

c
is an integer, a REAL*4, or a REAL*8 constant. (One of the pair must be a REAL*8 constant.)

The parentheses and the comma are part of the constant and are required. See Section 2.3.3 for the rules governing the formation of REAL*8 constants.

A COMPLEX*16 constant occupies 16 bytes of VAX-11 storage and is interpreted as a complex number.

Examples of valid and invalid COMPLEX*16 constants follow:

Valid	Invalid
(1.7039D0,-1.7039D0)	(1.23D0) (second constant missing)
(+12739D3,0.D0)	(0.8Q0,0.4Q0) (REAL*16 constants not allowed)
	(1.0D300,-1.0D300) (both constants out of range for D_floating implementation of REAL*8; valid for G_floating implementation of REAL*8)

2.3.7 Octal and Hexadecimal Constants

Octal and hexadecimal constants are alternative ways to represent numeric constants. You can use them wherever numeric constants are allowed.

An octal constant is a string of octal digits enclosed by apostrophes and followed by the alphabetic character O. An octal constant has the form

$'c_1c_2c_3...c_n'O$

where:

c
is a digit in the range 0 to 7.

A hexadecimal constant is a string of hexadecimal digits enclosed by apostrophes and followed by the alphabetic character X. A hexadecimal constant has the form

$'c_1c_2c_3...c_n'X$

where:

c
is a hexadecimal digit in the range 0 to 9, or a letter in the range A to F or a to f.

Leading zeros are ignored in octal and hexadecimal constants. You can specify up to 128 bits (43 octal digits, 32 hexadecimal digits).

Examples of valid and invalid octal constants are:

Valid	Invalid	
'07737'O	'7782'O	(invalid character)
'1'O	7772'O	(no initial apostrophe)
	'0737'	(no O after second apostrophe)

Examples of valid and invalid hexadecimal constants are:

Valid	Invalid	
'AF9730'X	'999.'X	(invalid character)
'FFABC'X	'F9X	(no apostrophe before the X)

Octal and hexadecimal constants are typeless numeric constants. They assume data types based on the way they are used (and thus are not converted before use), as follows:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
INTEGER*2 ICOUNT REAL*8 DOUBLE		
RAPHA = '99AF2'X	REAL*4	4
JCOUNT = ICOUNT + '777'O	INTEGER*2	2
DOUBLE = 'FFF99A'X	REAL*8	8
IF(N.EQ.'123'O) GO TO 10	INTEGER*4	4

- When a specific data type --generally integer-- is required, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
Y(IX)=Y('15'O)+3.	INTEGER*4	4

- When the constant is used as an actual argument, no data type is assumed; however, a length of four bytes is always used. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC('34BC2'X)	none	4

- When the constant is used in any other context, INTEGER*4 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF('AF77'X) 1,2,3	INTEGER*4	4
I = '7777'O - 'A39'X	INTEGER*4	4
J = .NOT.'73777'O	INTEGER*4	4

An octal or hexadecimal constant actually specifies as much as 16 bytes of data. When the data type implies that the length of the constant is more than the number of digits specified, the leftmost digits have a value of zero. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left. An error results if any nonzero digits are truncated. Table 2-2 (in Section 2.2.1) lists the number of bytes that each data type requires.

2.3.8 Logical Constants

A logical constant specifies a logical value, true or false. Thus, only the following two logical constants are possible:

.TRUE.

.FALSE.

The delimiting periods are a required part of each constant.

2.3.9 Character Constants

A character constant is a string of printable ASCII characters enclosed by apostrophes.

A character constant has the form

$'c_1c_2c_3...c_n'$

where:

c
is a printable character.

Both delimiting apostrophes must be present.

The value of a character constant is the string of characters between the delimiting apostrophes. The value does not include the delimiting apostrophes, but does include all spaces or tabs within the apostrophes.

Within a character constant, the apostrophe character is represented by two consecutive apostrophes (with no space or other character between them).

The length of the character constant is the number of characters between the apostrophes, except that two consecutive apostrophes represent a single apostrophe. The length of a character constant must be in the range 1 to 2000.

Examples of valid and invalid character constants are:

Valid		Invalid
'WHAT?'	'HEADINGS	(no trailing apostrophe)
'TODAY''S DATE IS: '	''	(character constant must contain at least one character)
'HE SAID, "HELLO"'	"NOW OR NEVER"	(quotation marks cannot be used in place of apostrophes)

If a character constant appears in a numeric context (for example, as the expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant. See Section 2.3.10.

2.3.10 **Hollerith Constants**

A Hollerith constant is a string of printable characters preceded by a character count and the letter H.

A Hollerith constant has the form

$$nHc_1c_2c_3\ldots c_n$$

where:

n
is an unsigned, nonzero integer constant stating the number of characters in the string (including spaces and tabs).

c
is a printable character.

The maximum number of characters is 2000.

Hollerith constants are stored as byte strings, one character per byte.

Hollerith constants have no data type. They assume a numeric data type according to the context in which they are used. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

Examples of valid and invalid Hollerith constants are:

Valid	Invalid
16HTODAY'S DATE IS:	3HABCD (wrong number of characters)
1HB	

When Hollerith constants are used in numeric expressions, they assume data types according to the following rules:

- When the constant is used with a binary operator, including the assignment operator, the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant
INTEGER*2 ICOUNT REAL*8 DOUBLE		
RALPHA = 4HABCD	REAL*4	4
JCOUNT = ICOUNT + 2HXY	INTEGER*2	2
DOUBLE = 8HABCDEFGH	REAL*8	8
IF(N.EQ.1HZ) GO TO 10	INTEGER*4	4

- When a specific data type is required, generally integer, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant
Y(IX)=Y(1HA)+3.	INTEGER*4	4

- When the constant is used as an actual argument, no data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
CALL APAC (9HABCDEFGHI)	none	9

- When the constant is used in any other context, INTEGER*4 data type is assumed. For example:

Statement	Data Type of Constant	Length of Constant
IF (2HAB) 1,2,3	INTEGER*4	4
I= 1HC-1HA	INTEGER*4	4
J= .NOT. 1HB	INTEGER*4	4

When the length of the constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. An error results if any nonspace characters are truncated.

Table 2-2 (in Section 2.2.1) lists the number of characters required for each data type. Each character occupies one byte of storage.

2.4 VARIABLES

A variable is a symbolic name associated with a storage location. The value of the variable is the value currently stored in that location; that value can be changed by assigning a new value to the variable. (See Section 2.1 for the form of a symbolic name.)

Variables are classified by data type, just as constants are. The data type of a variable indicates the type of data it represents, its precision, and its storage requirements. When data of any type is assigned to a variable, it is converted, if necessary, to the data type of the variable. You can establish the data type of a variable by type declaration statements, IMPLICIT statements, or predefined typing rules.

Two or more variables are associated with each other when each is associated with the same storage location. They are partially associated when part (but not all) of the storage associated with one variable is the same as part or all of the storage associated with another variable. Association and partial association occur when you use COMMON statements, EQUIVALENCE statements, or actual arguments and dummy arguments in subprogram references.

A variable is considered defined if the storage associated with it contains data of the same type as that of the name. A variable can be defined before program execution by a DATA statement or during execution by an assignment or input statement.

If variables of different data types are associated (or partially associated) with the same storage location, and the value of one variable is defined (for example, by assignment), the value of the other variable becomes undefined.

2.4.1 Data Type Specification

Type declaration statements (see Section 5.2) specify that given variables are to represent specified data types. For example:

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with an 8-byte storage location that is to contain complex data, and that the variable VAR2 is to be associated with an 8-byte double precision storage location.

The IMPLICIT statement (see Section 5.1) has a broader scope. It states that, in the absence of an explicit type declaration, any variable with a name that begins with a specified letter, or any letter within a specified range, is to represent a specified data type.

You can explicitly specify the data type of a variable only once. An explicit data type specification takes precedence over the type implied by an IMPLICIT statement.

Character type declaration statements (see Sections 5.1 and 5.2.2) specify that given variables are to represent character values with the length specified. For example:

```
CHARACTER*72 INLINE
CHARACTER NAME*12, NUMBER*9
```

These statements indicate that the variables INLINE, NAME, and NUSMBER are to be associated with storage locations containing character data of lengths 72, 12, and 9, respectively.

Passed length character arguments are used within a single subprogram to process character strings of different lengths. The passed length character argument has a length specification of (*). For example:

```
CHARACTER*(*) CHARDUMMY
```

The passed length character argument assumes the length of the actual argument (see Chapter 6).

2.4.2 Data Type by Implication

In the absence of either IMPLICIT statements or explicit type statements, all variables with names beginning with I, J, K, L, M, or N are assumed to be integer variables. Variables with names beginning with any other letter are assumed to be real variables. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM
TOTAL	NTOTAL

2.5 ARRAYS

An array is a group of contiguous storage locations associated with a single symbolic name, the array name. The individual storage locations, called array elements, are referred to by a subscript appended to the array name. Section 2.5.2 discusses subscripts.

An array can have from one to seven dimensions. For example, a column of figures is a one-dimensional array. A table of more than one column of figures is a two-dimensional array. To refer to a specific value in this array, you must specify both its row number and its column number. A table of figures that covers several pages is a three-dimensional array. To locate a value in this array, you must specify the row number, column number, and a page number.

The following FORTRAN statements establish arrays:

- Type declaration statements (see Section 5.2)
- The DIMENSION statement (see Section 5.3)
- The COMMON statement (see Section 5.4)

These statements contain array declarators (see Section 2.5.1) that define the name of the array, the number of dimensions in the array, and the number of elements in each dimension.

An element of an array is considered defined if the storage associated with it contains data of the same data type as that of the array name (see Section 2.5.4). You can define an array element or an entire array before program execution with a DATA statement. During program execution, you can define an array element with an assignment or input statement, and an entire array with an input statement.

2.5.1 Array Declarators

An array declarator specifies the symbolic name that identifies an array within a program unit and indicates the properties of that array.

An array declarator has the form

```
a (d[,d] ...)
```

where:

a
is the symbolic name of the array, that is, the array name. (Section 2.1 gives the form of a symbolic name.)

d
is a dimension declarator; d can specify both a lower bound and an upper bound as follows:

```
[dl:]du
```

dl
is the lower bound of the dimension.

du
is the upper bound of the dimension. (A * can also occur as an upper bound, but only of the last dimension.)

The number of dimension declarators indicates the number of dimensions in the array. The number of dimensions can range from one to seven.

The value of the lower bound dimension declarator can be negative, zero, or positive. The value of the upper bound dimension declarator must be greater than or equal to that of the corresponding lower bound dimension declarator. The number of elements in the dimension is du-dl+1. If a lower bound is not specified, it is assumed to be 1, and the value of the upper bound specifies the number of elements in that dimension. For example, a dimension declarator of 50 indicates that the dimension contains 50 elements. The upper bound in the last dimension declarator in a list of dimension declarators may be an asterisk; an asterisk marks the declarator as an assumed-size array declarator (see Section 6.1.1.2).

Each dimension bound is an integer arithmetic expression in which each operand is an integer constant, an integer dummy argument, or an integer variable in a COMMON block.

Note that array references and function references are not allowed in dimension bounds expressions.

Dimension bounds that are not constant expressions can be used in a subprogram to define adjustable arrays. You can use adjustable arrays within a single subprogram to process arrays with different dimension bounds by specifying the array name as a subprogram argument, and by either specifying the bounds as subprogram arguments or placing the bounds in a COMMON block. See Section 6.1.1.1 for more information on adjustable arrays. Dimension bounds that are not constant expressions are not permitted in a main program.

The number of elements in an array is equal to the product of the number of elements in each dimension.

An array name can appear in only one array declarator within a program unit.

2.5.2 Subscripts

A subscript qualifies an array name. A subscript is a list of expressions, called subscript expressions, enclosed in parentheses, that determine which element in the array is referred to. The subscript is appended to the array name it qualifies.

A subscript has the form

```
(s[,s]...)
```

where:

s
is a subscript expression.

A subscripted array reference must contain one subscript expression for each dimension defined for that array (one for each dimension declarator).

Each subscript can be any valid arithmetic expression. If the value of a subscript expression is not of type integer, it is converted before use to an integer value by truncating of any fractional part.

2.5.3 Array Storage

As discussed earlier in this section, you can think of the dimensions of an array as rows, columns, and levels or planes. However, FORTRAN always stores an array in memory as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored so that the leftmost subscripts vary most rapidly. This is called the "order of subscript progression." For example, Figure 2-1 shows array storage in one, two, and three dimensions.

2.5.4 Data Type of an Array

The data type of an array is specified in the same way as the data type of a variable -- that is, the data type of an array is specified implicitly by the initial letter of the name, or explicitly by a type declaration statement.

All the values in an array have the same data type. Any value assigned to an array element is converted to the data type of the array. If an array is named in a DOUBLE PRECISION statement, for example, the compiler allocates an 8-byte storage location for each element of the array. When a value of any type is assigned to any element of that array, it is converted to double precision.

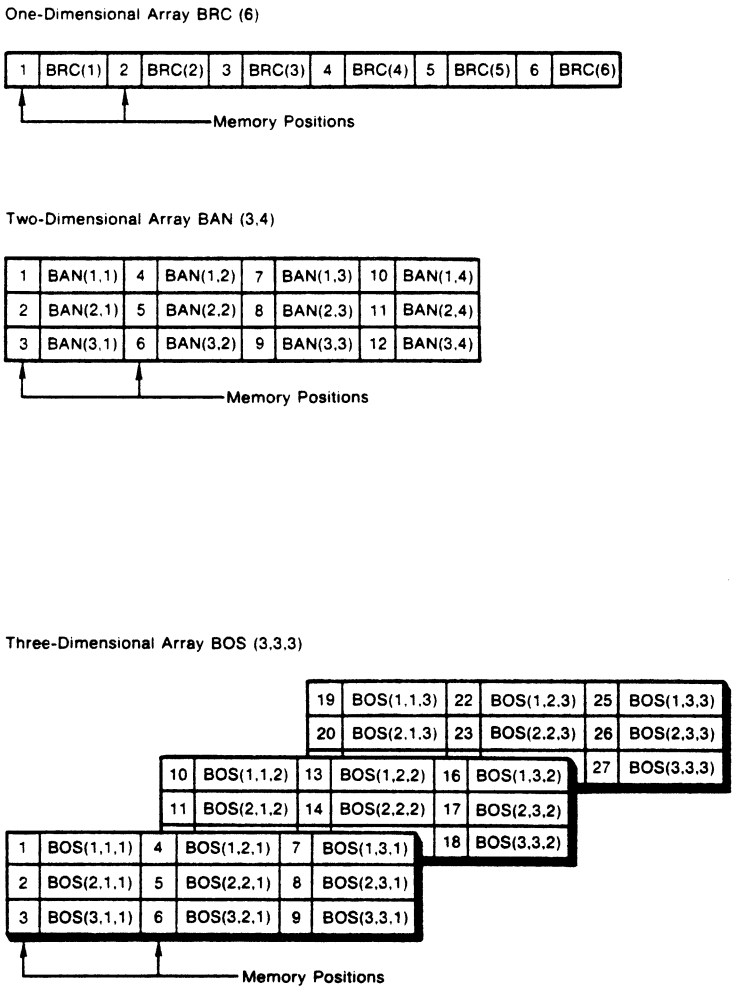


Figure 2-1 Array Storage

2.5.5 Array References Without Subscripts

In the following types of statements, you can specify an array name without a subscript to indicate that the entire array is to be used (or defined):

- Type declaration statements
- COMMON statement
- DATA statement
- EQUIVALENCE statement
- FUNCTION statement
- SUBROUTINE statement

- ENTRY statement
- SAVE statement
- Input/output statements

You can also use unsubscripted array names as actual arguments in references to external procedures. The use of unsubscripted array names in all other types of statements is not permitted.

2.5.6 Adjustable Arrays

Adjustable arrays allow subprograms to manipulate arrays of variable dimensions. To use an adjustable array in a subprogram, you specify the array bounds, as well as the array's name, as subprogram arguments. (The bounds may also be given in a COMMON block.) See Chapter 6 for more information.

2.6 CHARACTER SUBSTRINGS

A character substring is a contiguous segment of a character variable or character array element.

A character substring reference has one of the following forms:

```
v([e1]:[e2])

a(s[,s]...) ([e1]:[e2])
```

where:

v
is a character variable name.

a
is a character array name.

s
is a subscript expression.

e1
is a numeric expression that specifies the leftmost character position of the substring.

e2
is a numeric expression that specifies the rightmost character position of the substring.

Character positions within a character variable or array element are numbered from left to right, beginning at 1. For example, LABEL(2:7) specifies the substring beginning with the second character position and ending with the seventh character position of the character variable LABEL. If the CHARACTER*8 variable LABEL has a value of XVERSUSY, then the substring LABEL(2:7) has a value of VERSUS.

If the value of the numeric expression e1 or e2 is not of type integer, it is converted to an integer value by truncating any fractional part before use.

The values of the numeric expressions e1 and e2 must meet the following conditions:

```
1  .LE.  e1  .LE.  e2  .LE.  len
```

where:

len
is the length of the character variable or array element.

If e1 is omitted, FORTRAN assumes that e1 equals 1. If e2 is omitted, FORTRAN assumes that e2 equals len.

For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

2.7 EXPRESSIONS

An expression represents a single value. It can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed, using the values of the basic components, to obtain a single value.

Expressions are classified as arithmetic, character, relational, or logical. Arithmetic expressions produce numeric values; character expressions produce character values; and relational and logical expressions produce logical values.

2.7.1 Arithmetic Expressions

Arithmetic expressions are formed with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic element can be any of the following:

- A numeric, Hollerith, octal, or hexadecimal constant
- A numeric variable
- A numeric array element
- An arithmetic expression enclosed in parentheses
- An arithmetic function reference

The term "numeric," as used above, includes logical data, because logical data is treated as integer data when used in an arithmetic context.

Arithmetic operators specify a computation to be performed using the values of arithmetic elements. They produce a numeric value as a result. The operators and their meanings are:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and unary plus
-	Subtraction and unary minus

These operators are called binary operators because each is used with two elements. The plus (+) and minus (-) symbols are also unary operators when written immediately preceding an arithmetic element to denote a positive or negative value.

A variable or array element must have a defined value before it can be used in an arithmetic expression.

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. The precedence of the operators is:

Operator	Precedence
**	First
* and /	Second
+ and -	Third

When two or more operators of equal precedence (such as + and -) appear, they can be evaluated in any order, as long as the order of evaluation is algebraically equivalent to a left-to-right order of evaluation. Exponentiation, however, is evaluated from right to left. For example, A**B**C is evaluated as A** (B**C); B**C is evaluated first, and then A is raised to the resulting power.

2.7.1.1 **Use of Parentheses** - You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first, and the resulting value is used in the evaluation of the remainder of the expression. In the following examples, the numbers below the operators indicate the order of the evaluations:

```
4 + 3 * 2 - 6 / 2 = 7
  ↑   ↑   ↑   ↑
  2   1   4   3

(4+3) * 2 - 6 / 2 = 11
  ↑   ↑   ↑   ↑
  1   2   4   3

(4 + 3 * 2 - 6) / 2 = 2
   ↑   ↑   ↑     ↑
   2   1   3     4

((4+3) * 2 - 6) / 2 = 4
   ↑   ↑   ↑     ↑
   1   2   3     4
```

As shown in the third and fourth examples above, expressions within parentheses are evaluated according to the normal order of precedence, unless you override the order by using parentheses within parentheses.

Nonessential parentheses, as in the following expression, do not affect expression evaluation:

```
4 + (3*2) - (6/2)
```

The use of parentheses to specify the evaluation order is often important in high-accuracy numerical computations. In such computations, evaluation orders that are algebraically equivalent might not be computationally equivalent when processed by a computer.

2.7.1.2 **Data Type of an Arithmetic Expression** - If every element in an arithmetic expression is of the same data type, the value produced by the expression is also of that data type. If elements of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value depend on a rank associated with each data type. The rank assigned to each data type is as follows:

Data Type	Rank
Logical	1 (Low)
INTEGER*2	2
INTEGER*4	3
REAL*4 (REAL)	4
REAL*8 (DOUBLE PRECISION)	5
REAL*16	6
COMPLEX*8 (COMPLEX)	7
COMPLEX*16 (DOUBLE COMPLEX)	8 (High)

The data type of the value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. For example, the data type of the value resulting from an operation on an integer and a real element is real. However, an operation involving a COMPLEX*8 data type and either a REAL*8 or REAL*16 data type produces a COMPLEX*16 result.

The data type of an expression is the data type of the result of the last operation in that expression. The data type of an expression is determined by:

- Integer operations -- Integer operations are performed only on integer elements. (Logical entities used in an arithmetic context are treated as integers.) In integer arithmetic, any fraction that can result from division is truncated, not rounded. For example:

  ```
  1/4 + 1/4 + 1/4 + 1/4
  ```

  The value of this expression is 0, not 1.

- Real operations -- Real operations are performed only on real elements or combinations of real, integer, and logical elements. Any integer elements present are converted to real data type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. Note, however, that in the statement Y = (I/J)*X, an integer division operation is performed on I and J and a real multiplication is performed on that result and X.

- REAL*8 and REAL*16 operations -- Any element in an operation in which there is a higher-precision element is converted to the data type of the higher-precision element by making the existing element the most significant portion of the higher-precision datum. The least significant portion of the binary representation is zero. The expression is then evaluated in the higher-precision arithmetic.

- Converting a real element to a higher-precision element does not increase its accuracy. For example, a REAL variable having the value

  ```
  0.3333333
  ```

  is converted to (approximately)

  ```
  0.3333333134651184D0
  ```

  not to either

  ```
  0.3333333000000000D0
  ```

  or

  ```
  0.3333333333333333D0
  ```

- Complex operations -- In an operation that contains any complex element, integer elements are converted to real data type, as previously described. The REAL or REAL*8 element thus obtained is then designated as the real part of a complex number; the imaginary part is assigned a value of 0. The expression is then evaluated using complex arithmetic and the resulting value is of complex data type. Operations involving COMPLEX*8 and REAL*8 elements are done as COMPLEX*16 operations -- that is, the REAL*8 element is not rounded.

These rules also generally apply to arithmetic operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a REAL*8 or REAL*16 representation of the constant had been given. For example, the expression

```
1.0D0 + 0.3333333
```

is treated exactly as if it were

```
1.0D0 + 0.3333333000000000D0
```

2.7.2 Character Expressions

Character expressions consist of character elements and character operators. The evaluation of a character expression yields a single value of character data type.

A character element can be any one of the following:

- A character constant
- A character variable
- A character array element
- A character substring
- A character expression, optionally enclosed in parentheses
- A character function reference

The only character operator is the concatenation operator (//).

A character expression has the form

```
character element [//character element]...
```

The value of a character expression is a character string formed by successive left-to-right concatenations of the values of the elements of the character expression. The length of a character expression is the sum of the lengths of the character elements. For example, the value of the character expression 'AB'//'CDE' is 'ABCDE', which has a length of 5.

Parentheses do not affect the value of a character expression. For example, the following character expressions are equivalent:

```
('ABC'//'DE')//'F'
'ABC'//('DE'//'F')
'ABC'//'DE'//'F'
```

Each of these character expressions has the value 'ABCDEF'.

If a character element in a character expression contains spaces, the spaces are included in the value of the character expression. For example, 'ABC '//'D E'//'F ' has a value of 'ABC D EF '.

2.7.3 Relational Expressions

A relational expression consists of two arithmetic expressions or two character expressions, separated by a relational operator. The value of the expression is either true or false, depending on whether the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions or between two character expressions. These operators are:

Operator	Relationship
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The delimiting periods are a required part of each operator.

Complex expressions can be related only by the .EQ. and .NE. operators. Complex entities are equal if their corresponding real and imaginary parts are both equal.

In an arithmetic relational expression, the arithmetic expressions are first evaluated to obtain their values. These values are then compared to determine whether the relationship stated by the operator exists. For example:

```
APPLE+PEACH .GT. PEAR+ORANGE
```

This expression states the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If that relationship exists, the value of the expression is true; if not, the value of the expression is false.

In a character relational expression, the character expressions are first evaluated to obtain their values. These values are then compared to determine whether the relationship stated by the operator exists. In character relational expressions "less than" means "precedes in the ASCII collating sequence," and "greater than" means "follows in the ASCII collating sequence." For example:

```
'AB'//'ZZZ' .LT.  'CCCCC'
```

This expression states that 'ABZZZ' is less than 'CCCCC'. Since that relationship does exist, the value of the expression is true. If the relationship stated does not exist, the value of the expression is false.

If the two character expressions in a relational expression are not the same length, the shorter one is padded on the right with spaces until the lengths are equal. For example:

```
'ABC' .EQ.  'ABC   '

'AB' .LT.  'C'
```

The first relational expression has a value of true even though the lengths of the expressions are not equal, and the second has a value of true even though 'AB' is longer than 'C'.

All relational operators have the same precedence. However, arithmetic and character operators have a higher precedence than that of relational operators.

You can use parentheses, as in any other expression, to alter the order of evaluation of the expressions in a relational expression. However, because arithmetic and character operators are evaluated before relational operators, you need not enclose the entire arithmetic or character expression in parentheses.

A relational expression can compare two numeric expressions of different data types. In this case, the value of the expression with the lower-ranked data type is converted to the higher-ranked data type before the comparison is made.

2.7.4 Logical Expressions

A logical expression can be a single logical element, or a combination of logical elements and logical operators. A logical expression yields a single logical value, true or false.

A logical element can be any of the following:

- An integer or logical constant
- An integer or logical variable
- An integer or logical array element
- A relational expression
- A logical expression enclosed in parentheses
- An integer or logical function reference

The logical operators are:

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR): The expression is true if either A or B, or both, is true.
.XOR.	A .XOR. B	Logical exclusive OR: The expression is true if A is true and B is false, or vice versa; but the expression is false if both elements have the same value.
.NEQV.	A .NEQV. B	Same as .XOR.
.EQV.	A .EQV. B	Logical equivalence: The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation: The expression is true if, and only if, A is false.

The delimiting periods of logical operators are required.

When a logical operator operates on logical elements, the resulting data type is logical. When a logical operator operates on integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting data type is integer. When a logical operator combines integer and logical values, the logical value is first converted to an integer value; then the operation is carried out as for two integer elements. The resulting data type is integer.

A logical expression is evaluated according to an order of precedence assigned to its operators. Some logical expressions can be evaluated before all their subexpressions are evaluated. For example, if A is .FALSE., the expression A .AND. (F(X,Y) .GT. 2.0) .AND. B is .FALSE.. The value of the expression can be determined by testing A without evaluating F(X,Y). Thus, the function subprogram F may not be called, and side-effects resulting from the call -- for example, changing variables in COMMON -- will not occur.

The following list summarizes all the operators that can appear in a logical expression, in the order in which they are evaluated:

Operator	Precedence
**	First (Highest)
*,/	Second
+,-,//	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.,.NEQV.	Eighth

Operators of equal rank are evaluated from left to right. For example:

```
A*B+C*ABC .EQ. X*Y+DM/ZZ .AND. .NOT. K*B .GT. TT
```

The sequence in which this logical expression is evaluated is:

```
(((A*B)+(C*ABC)).EQ.((X*Y)+(DM/ZZ))).AND.(.NOT.((K*B).GT.TT))
```

As in arithmetic expressions, you can use parentheses to alter the normal sequence of evaluation.

Two logical operators cannot appear consecutively, unless the second operator is .NOT..

CHAPTER 3

ASSIGNMENT STATEMENTS

Assignment statements define the value of a variable, array element, or character substring. They do this by evaluating an expression and assigning the resulting value to the variable, array element, or character substring.

The four kinds of assignment statements are:

- Arithmetic
- Logical
- Character
- ASSIGN

3.1 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement assigns the value of the expression on the right of the equal sign to the numeric variable or array element on the left of the equal sign.

The arithmetic assignment statement has the form

```
v = e
```

where:

v
is a numeric variable or array element.

e
is an arithmetic expression.

The equal sign does not mean "is equal to," as in mathematics. It means "is replaced by." For example:

```
KOUNT = KOUNT + 1
```

This statement means, "replace the current value of the integer variable KOUNT with the sum of that current value and the integer constant 1."

Although the symbolic name on the left of the equal sign can be undefined, values must have been previously assigned to all symbolic references in the expression on the right of the equal sign.

The expression must yield a value that conforms to the requirements of the variable or array element to which it is to be assigned. For example, a real expression that produces a value greater than 32767 is invalid if the entity on the left of the equal sign is an INTEGER*2 variable. Significance may be lost if an INTEGER*4 value, which can exactly represent values of approximately the range -2*10**9 to +2*10**9, is converted to REAL*4 (including the real part of a complex constant), which is accurate to only about 7 digits.

If the variable or array element on the left of the equal sign has the same data type as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the entity on the left of the equal sign before it is assigned. Table 3-1 summarizes the data conversion rules for assignment statements.

Table 3-1
Conversion Rules for Assignment Statements

Variable or Array Element (V)	Expression (E)					
	Integer or Logical	REAL	REAL*8	REAL*16	COMPLEX	COMPLEX*16
Integer or Logical	Assign E to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate E to integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used	Truncate real part of E to integer and assign to V; imaginary part of E is not used
REAL	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS* portion of E to V; LS* portion of E is rounded	Assign MS* portion of E to V; LS* portion of E is rounded	Assign real part of E to V; imaginary part of E is not used	Assign MS* portion of the real part of E to V; LS* portion of the real part of E is rounded; imaginary part of E is not used
REAL*8	Append fraction (.0) to E and assign to V	Assign E to MS* portion of V; LS* portion of V is 0	Assign E to V	Same as above	Assign real part of E to MS* of V; LS* portion of V is 0; imaginary part of E is not used	Assign real part of E to V; imaginary part of E is not used
REAL*16	Same as above	Same as above	Assign E to MS* portion of V; LS* portion of V is 0	Assign E to V	Same as above	Assign real part of E to MS* portion of V;LS* portion of real part of V is 0.Imaginary part of E is not used
COMPLEX	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part of V is 0.0	Assign MS* portion of E to real part of V; LS* portion of E is rounded; imaginary part of V is 0.0	Assign MS* portion of E to real part of V;LS* portion of E is rounded; imaginary part of V is 0.0	Assign E to V	Assign MS* portion of real part of E to real part of V;LS* portion of real part of E is rounded. Assign MS* portion of imaginary part of E to imaginary part of V;LS* portion of imaginary part of E is rounded.
COMPLEX*16	Append fraction (.0) to E and assign to V;imaginary part of V is 0.0	Assign E to MS* portion of real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part is 0.0	Same as above	Assign real part of E to MS* portion of real part of V; LS* portion of real part is 0. Assign imaginary part of E to MS* portion of imaginary part of V; LS* portion of imaginary part is 0.	Assign E to V

*MS = most significant (high order)
LS = least significant (low order)

Examples of valid and invalid assignment statements are:

Valid

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.
```

Invalid

```
3.14 = A-B                  (entity on the left must be a
                             variable or array element)

-J = I**4                   (entity on the left must not be
                             signed)

ALPHA = ((X+6)*B*B/(X-Y)    (left and right parentheses do not
                             balance)

ICOUNT = A//B(3:7)          (expression on the right must not be of
                             character data type if the entity on
                             the left is not of character data type)
```

3.2 LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement assigns the value of the logical expression on the right of the equal sign to the variable or array element on the left of the equal sign. See Table 3-1 for conversion rules.

The logical assignment statement has the form

```
v = e
```

where:

v
is a logical variable or array element.

e
is a logical expression.

The variable or array element on the left of the equal sign must be of logical data type. Values must have been previously assigned to all symbolic references that appear in the expression. The expression must yield a logical value.

Examples of logical assignment statements are:

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

3.3 CHARACTER ASSIGNMENT STATEMENT

The character assignment statement assigns the value of the character expression on the right of the equal sign to the character variable, array element, or substring on the left of the equal sign.

The character assignment statement has the form

```
v = e
```

where:

v
is a character variable, array element, or substring.

e
is a character expression.

If the length of the character expression is greater than the length of the character variable, array element, or substring, the character expression is truncated on the right.

If the length of the character expression is less than the length of the character variable, array element, or substring, the character expression is filled on the right with spaces.

The expression must be of character data type. You cannot assign a numeric value to a character variable, array element, or substring.

Note that assigning a value to a character substring does not affect character positions in the character variable or array element not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged; and if the character position is undefined, it remains undefined.

Examples of valid and invalid character assignment statements follow. Note that all variables and arrays in the examples are assumed to be of character data type.

Valid

```
FILE = 'PROG2'

REVOL(1) = 'MAR'//'CIA'

LOCA(3:8) = 'PLANT5'

TEXT(I,J+1)(2:N-1) = NAME//X
```

Invalid

```
'ABC' = CHARS       (element on the left must be a character
                     variable, array element, or substring
                     reference)

CHARS = 25          (expression on the right must be of character
                     data type)
```

ASSIGN

3.4 ASSIGN STATEMENT

The ASSIGN statement assigns a statement label value to an integer variable. The variable can then be used as a transfer destination in a subsequent assigned GO TO statement, or as a format specifier in a formatted I/O statement.

The ASSIGN statement has the form

```
ASSIGN s TO v
```

where:

s
is the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

v
is an integer variable.

The ASSIGN statement assigns the statement number to the variable. It is similar to an arithmetic assignment statement, with one exception: The variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

The ASSIGN statement must be executed before the statement(s) in which the assigned variable is to be used. Moreover, the ASSIGN statement and the statement(s) in which the assigned variable is used must occur in the same program unit. For example:

```
ASSIGN 100 TO NUMBER
```

This statement associates the variable NUMBER with the statement label 100. Arithmetic operations on the variable, as in the following statement, then become invalid because arithmetic on label values is undefined:

```
NUMBER = NUMBER+1
```

The next statement dissociates NUMBER from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable:

```
NUMBER = 10
```

The variable NUMBER can no longer be used in an assigned GO TO statement.

Examples of ASSIGN statements are:

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR          (ERROR must have been defined as an
                              integer variable)
```

CHAPTER 4

CONTROL STATEMENTS

Statements are normally executed in the order in which they are written. However, you may cause an interruption of normal program flow in order to transfer control to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow; or it may be based on a decision made at that point.

You use the FORTRAN control statements to transfer control to a point within the same program unit or to another program unit. These statements also govern iterative processing, suspension of program execution, and program termination.

The control statements are:

- GO TO statements -- transfer control within a program unit
- IF statements -- conditionally transfer control, or conditionally execute a statement
- IF-THEN, ELSE IF THEN, ELSE, and END IF statements -- conditionally execute blocks of statements
- DO statements -- specify repetitive processing
- END DO statement -- terminates DO and DO WHILE loops
- CONTINUE statement -- transfers control to the next executable statement
- CALL statement -- invokes a subroutine subprogram
- RETURN statement -- returns control from a subprogram to the calling program unit
- PAUSE statement -- temporarily suspends program execution
- STOP statement -- terminates program execution
- END statement -- marks the end of a program unit

The following sections describe these statements, giving their forms and examples of how they are used.

GOTO

4.1 GO TO STATEMENTS

GO TO statements transfer control within a program unit. Depending on the value of an expression, control is transferred either to the same statement every time GO TO is executed, or to one of a set of statements.

The three types of GO TO statement are:

- Unconditional GO TO statement
- Computed GO TO statement
- Assigned GO TO statement

4.1.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement every time it is executed.

The unconditional GO TO statement has the form

```
GO TO s
```

where:

s
is the label of an executable statement in the same program unit as that of the GO TO statement.

The unconditional GO TO statement transfers control to the statement identified by the specified label. The label must identify an executable statement in the same program unit as that of the GO TO statement.

Examples of GO TO statements are:

```
GO TO 7734

GO TO 99999
```

4.1.2 Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an expression within the statement.

The computed GO TO statement has the form

```
GO TO (slist)[,] e
```

where:

slist
is a list of one or more labels of executable statements separated by commas. The list of labels is called the transfer list.

e
is an arithmetic expression in the range 1 to n (where n is the number of statement labels in the transfer list).

The computed GO TO statement evaluates the expression e and, if necessary, converts the resulting value to integer data type. Control is transferred to the statement label in position e in the transfer list. For example, if the list contains (30,20,30,40), and the value of e is 2, control is transferred to statement 20.

If the value of e is less than 1, or greater than the number of labels in the transfer list, control is transferred to the first executable statement after the computed GO TO.

Examples of computed GO TO statements are:

```
GO TO (12,24,36),INDEX

GO TO (320,330,340,350,360), SITU(J,K)+1
```

4.1.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label that is represented by a variable. An ASSIGN statement must establish the relationship between the variable and a specific statement label. Thus, the transfer destination can be changed, depending on the most recently executed ASSIGN statement.

The assigned GO TO statement has the form

```
GO TO v[[,](slist)]
```

where:

v
is an integer variable.

slist
is a list of one or more labels of executable statements separated by commas; slist does not affect statement execution and can be omitted.

The assigned GO TO statement transfers control to the statement whose label was most recently assigned to the variable v. The variable v must be integer data type and must have been assigned a statement label value by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

The assigned GO TO statement and its associated ASSIGN statement(s) must exist in the same program unit. Also statements to which control is transferred must be in this program unit; these statements must be executable statements.

Examples of assigned GO TO statements are:

```
ASSIGN 200 TO IGO
GO TO IGO
Equivalent to GO TO 200.

ASSIGN 450 TO IBEG
GO TO IBEG, (300,450,1000,25)
Equivalent to GO TO 450.
```

IF

4.2 IF STATEMENTS

IF statements conditionally transfer control, or conditionally execute a statement or block of statements. The three types of IF statements are:

- Arithmetic IF
- Logical IF
- Block IF (IF THEN, ELSE IF THEN, ELSE, END IF)

For each type, the decision to transfer control or to execute the statement or block of statements is based on the evaluation of an expression within the IF statement.

4.2.1 Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three statements, based on the value of an arithmetic expression.

The arithmetic IF statement has the form

```
IF (e) s1, s2, s3
```

where:

e
is an arithmetic expression.

s1,s2,s3
are labels of executable statements in the same program unit.

All three labels (s1,s2,s3) are required; however, they need not refer to three different statements.

The arithmetic IF statement first evaluates the expression (e) in parentheses. It then transfers control to one of the three statement labels in the transfer list, as follows:

If the value is:	Control passes to:
Less than 0	Label s1
Equal to 0	Label s2
Greater than 0	Label s3

Examples of arithmetic IF statements follow.

```
IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (NUMBER/2*2-NUMBER) 20,40,20
```

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even; it transfers control to statement 20 if the value is odd.

4.2.2 Logical IF Statement

A logical IF statement conditionally executes a single FORTRAN statement. The decision to execute the statement is based on the value of a logical expression within the logical IF statement.

The logical IF statement has the form

```
IF (e) st
```

where:

e
is a logical expression.

st
is a complete FORTRAN statement. The statement can be any executable statement except a DO statement, an END DO statement, an END statement, a block IF statement, or another logical IF statement.

The logical IF statement first evaluates the logical expression (e). If the value of the expression is true, the statement (st) is executed. If the value of the expression is false, control transfers to the next executable statement after the logical IF. The statement (st) is not executed.

Examples of logical IF statements are:

```
IF (J .GT. 4 .OR. J .LT. 1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*(-1.5D0)

IF (ENDRUN) CALL EXIT
```

4.2.3 Block IF Statements

Block IF statements conditionally execute blocks (or groups) of statements.

The four block IF statements are:

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

These statements are used in block IF constructs. The block IF construct has the form

```
IF (e) THEN
   block

ELSE IF (e) THEN
   block
    .
    .
    .

ELSE
   block

END IF
```

where:

e
is a logical expression.

block
is a sequence of zero or more complete FORTRAN statements. This sequence is called a statement block.

Figure 4-1 describes the flow of control for four examples of block IF constructs.

Each block IF statement, except the END IF statement, has an associated statement block. The statement block consists of all the statements following the block IF statement up to (but not including) the next block IF statement in the block IF construct. The statement block is conditionally executed based on the values of logical expressions in the preceding block IF statements.

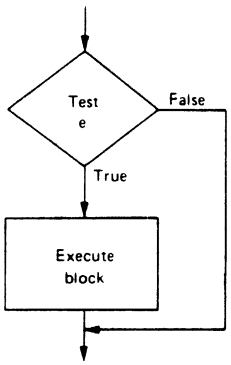
The IF THEN statement begins a block IF construct. The block following it is executed if the value of the logical expression in the IF THEN statement is true.

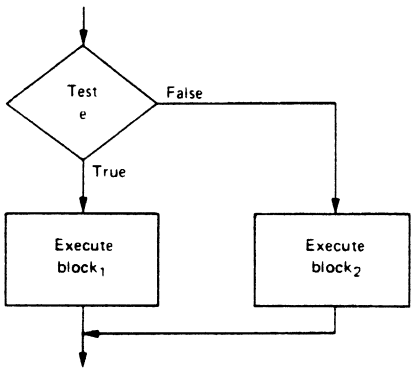
The ELSE IF THEN statement is an optional statement that specifies a statement block to be executed if the value of the logical expression in the ELSE IF THEN statement is true, and no preceding statement block in the block IF construct was executed. A block IF construct can contain any number of ELSE IF THEN statements.

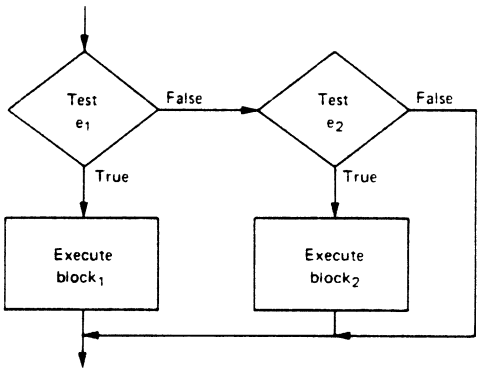
The ELSE statement specifies a statement block to be executed if no preceding statement block in the block IF construct was executed. Except for the END IF statement, no block IF statement can follow the ELSE statement. The ELSE statement is optional.

The END IF statement terminates the block IF construct.

Construct	Flow of Control

IF (e) THEN
block
END IF
	

IF (e) THEN
$block_1$
ELSE
$block_2$
END IF
	

IF (e_1) THEN
$block_1$
ELSE IF (e_2) THEN
$block_2$
END IF
	

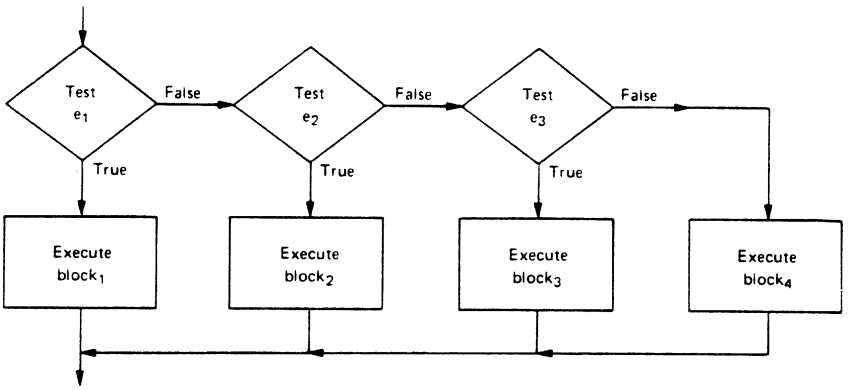
IF (e_1) THEN
$block_1$
ELSE IF (e_2) THEN
$block_2$
ELSE IF (e_3) THEN
$block_3$
ELSE
$block_4$
END IF
	

Figure 4-1 Examples of Block IF Constructs

After the last statement in a statement block is executed, control passes to the next executable statement following the END IF statement. Consequently, at most one statement block in a block IF construct is executed each time the IF THEN statement is executed.

ELSE IF THEN and ELSE statements can have statement labels, but these labels cannot be referenced. The END IF statement can have a statement label to which control can be transferred, but only from within the block IF construct.

Section 4.2.3.1 describes restrictions on statements in a statement block. Section 4.2.3.2 describes examples of block IF constructs. Section 4.2.3.3 describes nested block IF constructs.

4.2.3.1 **Statement Blocks** - A statement block can contain any executable FORTRAN statement except an END statement. You can transfer control out of a statement block but control cannot be transferred back into the block. Note that you cannot transfer control from one statement block to another.

DO loops cannot partially overlap statement blocks. When a statement block contains a DO statement, it must also contain the DO loop's terminal statement, and vice versa. If you use DO loops with statement blocks, each loop must be wholly contained within one statement block.

4.2.3.2 **Block IF Examples** - The simplest block IF construct consists of the IF THEN and END IF statements; this construct conditionally executes one statement block.

Form	**Example**

IF (e) THEN
  block
END IF
	IF (ABS(ADJU).GE.1.0E-6) THEN
	  TOTERR=TOTERR+ABS(ADJU)
	  QUEST=ADJU/FNDVAL
	END IF

The statement block consists of all the statements between the IF THEN and the END IF statements.

The IF THEN statement first evaluates the logical expression (e), ABS(ADJU).GE.1.0E-6. If the value of e is true, the statement block is executed. If the value of e is false, control transfers to the next executable statement after the END IF statement; the block is not executed.

The following example contains a block IF construct with an ELSE IF THEN statement:

Form	**Example**

IF (e1) THEN
  block1
	IF (A. GT. B) THEN
	  D = B
	  F = A - B

ELSE IF (e2) THEN
  block2
	ELSE IF (A .GT. B/2.) THEN
	  D = B/2.
	  F = A - B/2.

END IF	END IF

Block1 consists of all the statements between the IF THEN and the ELSE IF THEN statements; block2 consists of all the statements between the ELSE IF THEN and the END IF statements.

If A is greater than B, block1 is executed.

If A is not greater than B but A is greater than B/2, block2 is executed.

If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed; control transfers directly to the next executable statement after the END IF statement.

The following example contains a block IF construct with an ELSE statement:

Form	**Example**
IF (e) THEN	IF (NAME .LT. 'N') THEN
block1	IFRONT = IFRONT + 1
	FRLET(IFRONT)=NAME(1:2)
ELSE	ELSE
block2	IBACK=IBACK + 1
END IF	END IF

Block1 consists of all the statements between the IF THEN and ELSE statements; block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than 'N', block1 is executed.

If the value of NAME is greater than or equal to 'N', block2 is executed.

The following example contains a block IF construct with several ELSE IF THEN statements and an ELSE statement:

Form	**Example**
IF (e1) THEN	IF (A .GT. B) THEN
block1	D = B
	F = A - B
ELSE IF (e2) THEN	ELSE IF (A .GT. C) THEN
block2	D = C
	F = A - C
ELSE IF (e3) THEN	ELSE IF (A .GT. Z) THEN
block3	D = Z
	F = A - Z
ELSE	ELSE
block4	D = 0.0
	F = A
END IF	END IF

There are four statement blocks in this example. Each consists of all the statements between the block IF statements listed below.

Block	**Delimiting Block IF Statements**
block1	IF THEN and first ELSE IF THEN
block2	First ELSE IF THEN and second ELSE IF THEN

block3 Second ELSE IF THEN and ELSE

block4 ELSE and END IF

If A is greater than B, block1 is executed.

If A is not greater than B but is greater than C, block2 is executed.

If A is not greater than B or C but is greater than Z, block3 is executed.

If A is not greater than B, C, or Z, block4 is executed.

4.2.3.3 **Nested Block IF Constructs** - A block IF construct can be included in a statement block of another block IF construct. But the nested block IF construct must be completely contained within a statement block; it must not overlap statement blocks.

The following example contains a nested block IF construct:

Form

```
IF (e) THEN
                ( IF (e) THEN
                (
block1          (      ELSE
                (   blockb
                ( END IF
ELSE
   block2
END IF
```

Example

```
IF (A .LT. 100) THEN
   INRAN=INRAN + 1
   IF (ABS (A-AVG) .LE. 5.) THEN
      INAVG = INAVG + 1
   ELSE
      OUTAVG = OUTAVG + 1
   END IF
ELSE
   OUTRAN = OUTRAN + 1
END IF
```

If A is less than 100, block1 is executed. Block1 contains a nested block IF construct. If the absolute value of A minus AVG is less than or equal to 5, blocka is executed. If the absolute value of A minus AVG is greater than 5, blockb is executed.

If A is greater than or equal to 100, block2 is executed; the nested IF construct is not executed because it is not in block2.

DO

4.3 DO STATEMENT

The two types of DO statements are:

- Indexed DO (DO)
- Pretested indefinite DO (DO WHILE)

DO is discussed in Section 4.3.1, and DO WHILE in Section 4.3.2.

4.3.1 The Indexed DO Statement

The indexed DO, or DO, statement controls iterative processing; that is, the statements in its range are executed repeatedly a specified number of times.

The DO statement has the form

```
DO  [s[,]]   v=e1,e2[,e3]
```

where:

s
is the label of an executable statement. The statement must physically follow in the same program unit.

v
is an integer, or real variable.

e1,e2,e3
are arithmetic expressions.

The variable v is the control variable; e1, e2, and e3 are the initial, terminal, and increment parameters, respectively. If you omit the increment parameter, a default increment value of 1 is used.

The optional label that appears in the DO statement identifies the terminal statement of the DO loop. If no label appears in the DO statement, the DO loop must be terminated by the END DO statement discussed in Section 4.4. The terminal statement must not be one of the following statements:

- Unconditional or assigned GO TO statement
- Arithmetic IF statement
- Any block IF statement
- END statement
- RETURN statement
- DO statement

The range of the DO statement includes all the statements that follow the DO statement, up to and including the terminal statement or END DO.

The DO statement first evaluates the expressions e1, e2, and e3 to determine values for the initial, terminal, and increment parameters, repectively. The increment parameter (e3) cannot be zero. If necessary, the initial, terminal, and increment parameters are converted, before use, to the data type of the control variable. The value of the initial parameter is assigned to the control variable.

The number of executions of the DO range, called the iteration count, is given by:

$$[(e2 - e1 + e3)/e3]$$

where the notation [X] represents the largest integer whose magnitude does not exceed the magnitude of X and whose sign is the same as the sign of X.

If the iteration count is zero or negative, the body of the loop is not executed.

If the /NOF77 compiler qualifier is specified, and the iteration count is zero or negative, the body of the loop is executed once.

4.3.1.1 **DO Iteration Control** - After each iteration of the DO range, the following steps are executed:

1. The value of the increment parameter is algebraically added to the control variable.

2. The iteration count is decremented.

3. If the iteration count is greater than zero, control transfers to the first executable statement after the DO statement for another iteration of the range.

4. If the iteration count is zero, execution of the DO statement terminates. The final value of the control variable is the value determined by step 1.

Note that if the data type of the control variable is real, the number of iterations of the DO range might not be what is expected, because of rounding errors.

You can also terminate execution of a DO statement by using a statement within the range that transfers control outside the loop. The control variable of the DO statement remains defined with its current value.

When execution of a DO loop terminates, and other DO loops share its terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (see Section 4.3.1.2). If no other DO loop shares the terminal statement, or if this DO statement is outermost, control transfers to the first executable statement after the terminal statement.

You cannot alter the value of the control variable within the range of the DO statement. However, you can use the control variable for reference as a variable within the range.

You can modify the initial, terminal, and increment parameters within the loop without affecting the iteration count.

The range of a DO statement can contain other DO statements, as long as these nested DO loops meet certain requirements. Section 4.3.1.2 describes these requirements.

You can transfer control out of a DO loop, but not into a loop from elsewhere in the program. Exceptions to this rule are described in Sections 4.3.1.3 and 4.3.1.4.

Examples of DO iteration control follow.

```
DO 100 K=1,50,2
```

This statement specifies 25 iterations; K=49 during the final iteration, K=51 after the loop.

```
DO 350 J=50,-2,-2
```

This statement specifies 27 iterations; J=-2 during the final iteration, J=-4 after the loop.

```
DO 25 IVAR=1,5
```

This statement specifies 5 iterations; IVAR=5 during the final iteration, IVAR=6 after the loop.

```
DO NUMBER=5,40,4
```

This statement specifies 9 iterations; the terminating statement of the DO loop must be END DO.

```
DO 40 M=2.10
```

This is an invalid DO statement; it contains a decimal point instead of a comma. This example illustrates a common typing error. It is the valid arithmetic assignment statement:

```
DO40M = 2.10
```

4.3.1.2 **Nested DO Loops** - A DO loop can contain one or more complete DO loops. The range of an inner nested DO loop must lie completely within the range of the next outer loop. Nested loops can share a labelled terminal statement but not an unlabelled END DO statement.

Figure 4-2 illustrates nested loops.

4.3.1.3 **Control Transfers in DO Loops** - Within a nested DO loop, you can transfer control from an inner loop to an outer loop. However, a transfer from an outer loop to an inner loop is not permitted.

If two or more nested DO loops share the same terminal statement, you can transfer control to that statement only from within the range of the innermost loop. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

4.3.1.4 **Extended Range** - A DO loop has an extended range if it contains a control statement that transfers control out of the loop and if, after execution of one or more statements, another control statement returns control back into the loop. Thus, the range of the loop is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

The following rules govern the use of a DO statement extended range:

1. A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

2. The extended range of a DO statement must not change the control variable of the DO statement.

Figure 4-3 on page 4-15 illustrates valid and invalid extended range control transfers.

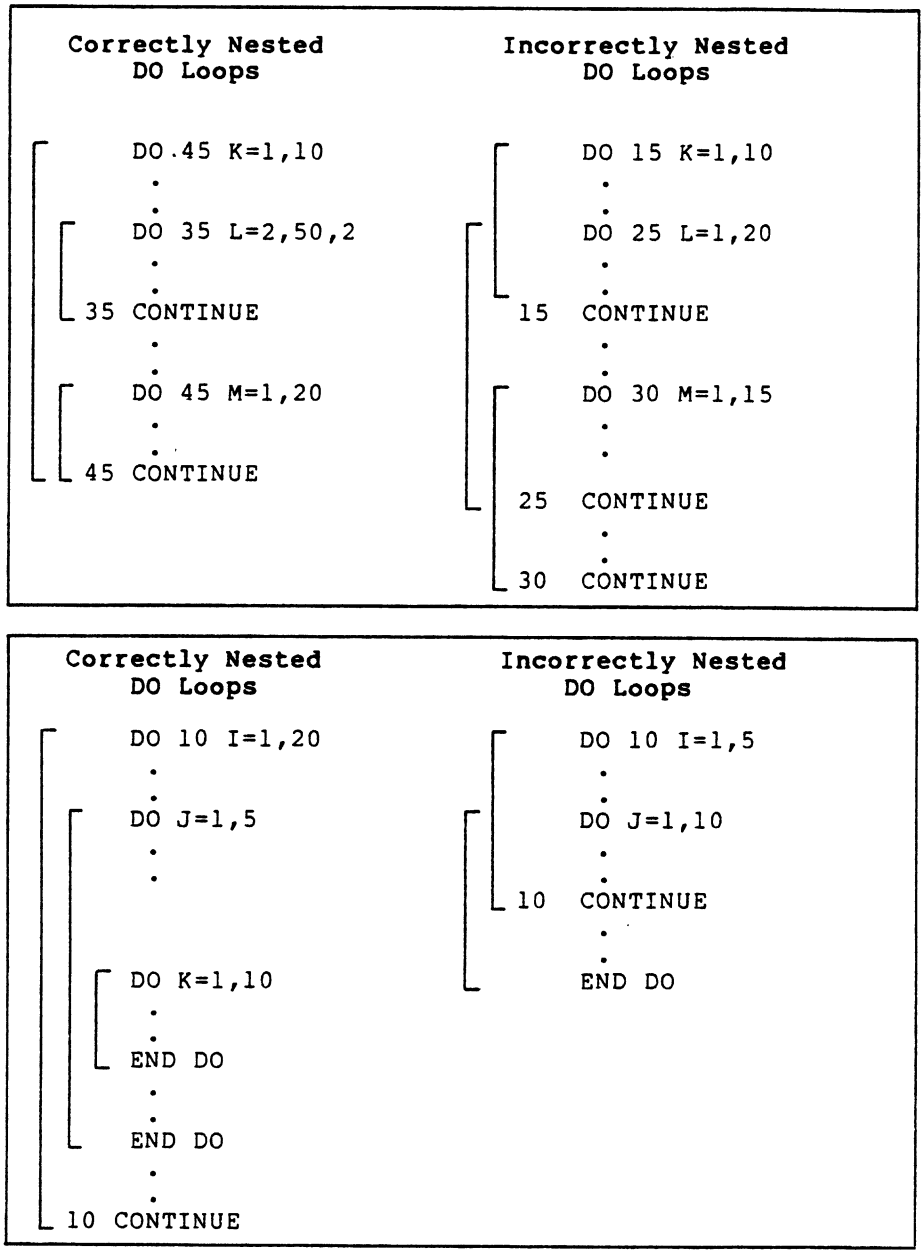


Figure 4-2 Nested DO Loops

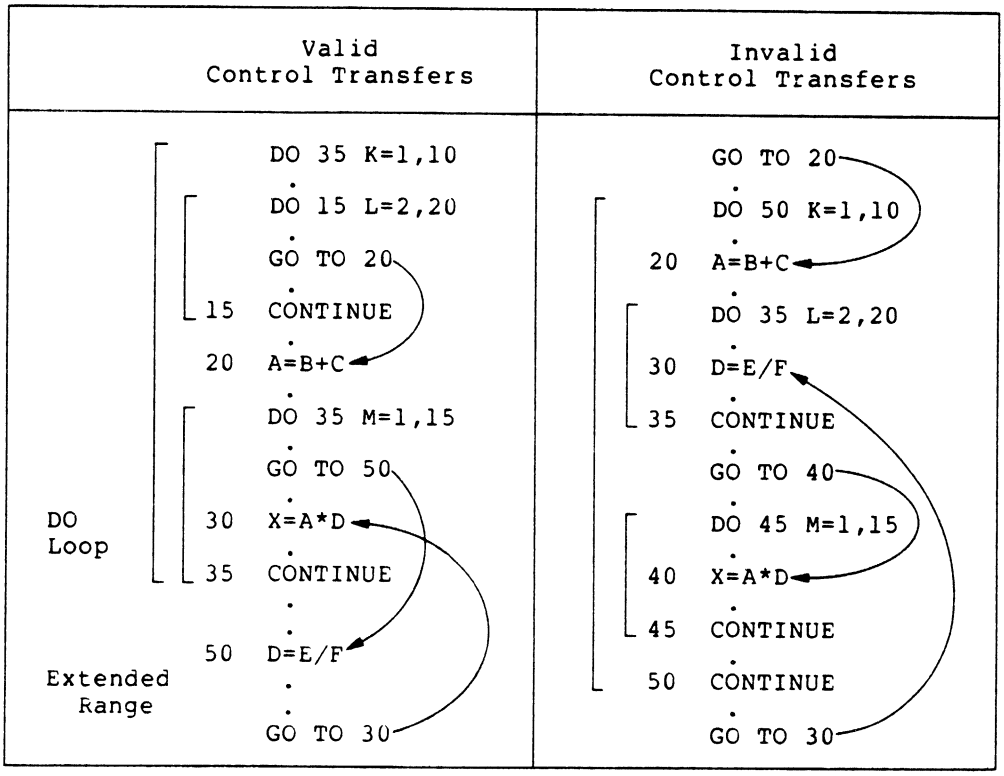


Figure 4-3 Control Transfers and Extended Range

DO WHILE

4.3.2 The DO WHILE Statement

The DO WHILE statement is similar to the DO statement discussed in the preceding section. Instead of executing for a set number of iterations, however, it executes for as long as a logical expression contained in the statement continues to be true.

The DO WHILE statement has the form

```
DO[s[,]] WHILE (e)
```

where:

s
is the label of an executable statement that must physically follow in the same program unit.

e
is a logical expression that can be tested to be either true or false.

The DO WHILE statement tests the logical expression at the beginning of each execution of the loop, including the first. If the value of the expression is true, the statements in the body of the loop are executed; if the expression is false, control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement (see Section 4.4).

The following example demonstrates the use of the DO WHILE statement:

```
CHARACTER*132 LINE
I=1
DO WHILE (LINE (I:I) .EQ.' ')
   I=I+1
END DO
```

See Section 4.4 for a discussion of the END DO statement used in the above example.

END DO

4.4 END DO STATEMENT

The END DO statement terminates the range of a DO or DO WHILE statement. The END DO statement must be used -- as an unlabelled terminal statement -- if the DO or DO WHILE statement defining the block does not contain a terminal-statement label; it may optionally be used -- as a labelled terminal statement -- if the DO or DO WHILE statement does contain a terminal-statement label.

The END DO statement has the form

```
END DO
```

Examples of the use of the END DO statement follow:

```
DO WHILE (I .GT. J)          DO 10 WHILE (I .GT. J)
    ARRAY (I,J)=1.0              ARRAY (I,J)=1.0
    I=I-1                        I=I-1
END DO                     10 END DO
```

CONTINUE

4.5 CONTINUE STATEMENT

The CONTINUE statement transfers control to the next executable statement. It is used primarily as the terminal statement of a labelled DO loop when that loop would otherwise end illegally with a GO TO, arithmetic IF, or other prohibited control statement.

The CONTINUE statement has the form

```
CONTINUE
```

CALL

4.6 CALL STATEMENT

The CALL statement executes a SUBROUTINE subprogram or other external procedure. It can also specify an argument list for the subroutine. (See Chapter 6 for greater detail on the definition and use of subroutines.)

The CALL statement has the form

```
CALL s[([a][,[a]]...)]
```

where:

s
is the name of a subroutine subprogram or other external procedure; or of a dummy argument associated with a subroutine subprogram or other external procedure.

a
is an actual argument. (Section 6.1 describes actual arguments.)

If you specify an argument list, the CALL statement associates the values in the list with the dummy arguments in the subroutine. It then transfers control to the first executable statement following the SUBROUTINE or ENTRY statement referenced by the CALL.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine. They can be variables, arrays, array elements, substring references, constants, expressions, Hollerith constants, alternate return specifiers, or subprogram names. An unsubscripted array name in the argument list refers to the entire array.

Examples of CALL statements are:

```
CALL CURVE (BASE,3.14159+X,Y,LIMIT,R(LT+2))

CALL PNTOUT (A,N,'ABCD')

CALL EXIT

CALL MULT (A,B,*10,*20,C)
```

The last example illustrates the use of statement label identifiers in CALL statement argument lists. The asterisk indicates that *10 and *20 are statement label identifiers. Label identifiers prefixed by asterisks (or ampersands (&)) are called alternate return specifiers (see Section 4.7).

RETURN

4.7 RETURN STATEMENT

The RETURN statement transfers control from a subprogram to the program that called the subprogram. It has the form

```
RETURN [i]
```

The optional argument is used to indicate an alternate return from the subprogram. It can be specified only in subroutine subprograms. When specified, the value of i indicates that the ith alternate return in the actual argument list is to be taken (see the second example below). The value of i can be any integer constant or expression, for example, 2 or I+J.

When a RETURN statement is executed in a function, control is returned to the calling program at the statement that contains the function reference (see Chapter 6). When a RETURN statement is executed in a subroutine, control is returned either to the first executable statement following the CALL statement that initiated the subroutine, or to the statement label that was specified as the ith alternate return in the CALL argument list.

You can use RETURN statements only in subprogram units. You cannot use the RETURN i form in function subprograms.

Examples of RETURN statements follow:

```
      SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
      INTEGER ALPH(*), DATA(*), PRNT(*)
      IF (N .GE. 10) THEN
           DATA(K+2) = N-(N/10)*N
           N = N/10
           DATA (K+1) = N
           PRNT (K+2) = ALPH(DATA(K+2)+1)
           PRNT (K+1) = ALPH(DATA(K+1)+1)
      ELSE
           PRNT(K+2) = ALPH(N+1)
      END IF
      RETURN
      END
```

In this example, control is returned to the calling program at the first executable statement following the CALL CONVRT statement.

```
      SUBROUTINE CHECK (X,Y,*,*,C)
                .
                .
                .
50    IF(Z) 60, 70, 80
60    RETURN
70    RETURN 1
80    RETURN 2
      END
```

This example shows how alternate returns can be included in a subroutine. If the value computed for Z is less than zero, a normal return is taken, and the calling program continues at the first executable statement following CALL CHECK. If Z equals zero, however, the first alternate return (RETURN 1) is taken; and if Z is greater than zero, the second alternate return (RETURN 2) is taken. Control is returned to the statement specified as the first or second alternate return argument in the CALL statement argument list. For example:

```
      CALL CHECK(A,B,&10,&20,C)
```

Thus, RETURN 1 transfers control to statement label 10, and RETURN 2 transfers control to statement label 20. Note that if a subroutine includes an alternate return that specifies a value either less than 1 or greater than the number of alternate return arguments, control is returned to the next executable statement after the CALL statement. That is, the alternate returns are ignored. Therefore, you should ensure that the value of i is within the range of alternate return arguments.

4.8 PAUSE STATEMENT

The PAUSE statement temporarily suspends program execution and displays a message on the terminal to permit you to take some action.

The PAUSE statement has the form

```
PAUSE  [disp]
```

where:

disp
is a character constant or a decimal digit string of one to five digits.

The disp argument is optional. The effect of a PAUSE statement depends on how your program is being executed. If it is running as a batch job or detached process, the contents of disp are written to the system output file but the program is not suspended.

If the program is running in interactive mode, the contents of disp are displayed at your terminal, followed by the prompt sequence, indicating that the program is suspended and that you should enter a command. For example, if the following statement is executed in interactive mode:

```
PAUSE 'ERRONEOUS RESULT DETECTED'
```

you will see the following display at the terminal:

```
ERRONEOUS RESULT DETECTED
$
```

If you do not specify a value for disp, you will receive the following message:

```
FORTRAN PAUSE
```

You can respond by typing one of the following commands:

CONTINUE - Execution resumes at the next executable statement.

STOP - Execution is terminated.

DEBUG - Execution resumes under control of the DEBUG program.

STOP

4.9 STOP STATEMENT

The STOP statement terminates program execution.

The STOP statement has the form

```
STOP  [disp]
```

where:

disp
is a character constant or a decimal digit string of one to five digits.

The disp argument is optional. If you specify it, the STOP statement displays the contents of disp at your terminal, terminates program execution, and returns control to the operating system. If you do not specify a value for disp, you will receive the following message:

```
FORTRAN STOP
```

Examples of STOP statements are:

```
STOP 98

STOP 'END OF RUN'
```

END

4.10 END STATEMENT

The END statement marks the end of a program unit. It must be the last source line of every program unit.

The END statement has the form

```
END
```

In a main program, if control reaches the END statement, program execution terminates. In a subprogram, a RETURN statement is implicitly executed.

CHAPTER 5

SPECIFICATION STATEMENTS

Specification statements are nonexecutable statements that let you allocate and initialize variables and arrays, and define other characteristics of the symbolic names used in the program.

The specification statements are:

- IMPLICIT statement -- overrides the implied data type of symbolic names.
- Type declaration statement -- explicitly defines the data type of specified symbolic names.
- DIMENSION statement -- defines the number of dimensions in an array and the number of elements in each dimension.
- COMMON statement -- defines one or more contiguous areas of storage.
- EQUIVALENCE statement -- associates two or more entities with the same storage location.
- SAVE statement -- retains values of local variables after a return from a subprogram.
- EXTERNAL statement -- allows use of user-supplied procedures as arguments to subprograms. (See Appendix A for a version of the EXTERNAL statement that is compatible with earlier versions of Digital FORTRAN.)
- INTRINSIC statement -- allows use of FORTRAN intrinsic functions as arguments to subprograms.
- DATA statement -- assigns initial values to variables, arrays, and array elements before program execution.
- PARAMETER statement -- assigns a symbolic name to a constant value. (See Appendix A for a version of the PARAMETER statement that is compatible with earlier versions of Digital FORTRAN.)
- PROGRAM statement -- assigns a symbolic name to a main program unit.
- BLOCK DATA statement -- establishes and defines common blocks and assigns initial values to entities contained in those common blocks.

The following sections detail these statements, giving their forms and examples of how they are used.

IMPLICIT

5.1 IMPLICIT STATEMENT

By default, all names beginning with the letters I through N are assumed to be integer data type, and all names beginning with any other letter are assumed to be REAL*4 data type. The IMPLICIT statement overrides implied data typing of symbolic names.

The IMPLICIT statement has the form

```
IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
```

where:

typ
is one of the data type specifiers. (See Chapter 2, Table 2-2.)

a
is an alphabetic specification in either of the general forms: c or c1-c2, where c is an alphabetic character. The latter form specifies a range of letters, from c1 through c2, which must occur in alphabetical order.

When you specify typ as CHARACTER*len, len specifies the length for character data type. Len is an unsigned integer constant or an integer constant expression enclosed in parentheses, and must be in the range 1 through 32767.

The IMPLICIT statement assigns the specified data type to all symbolic names that begin with any specified letter, or any letter in a specified range, and which have no explicit data type declaration. For example:

```
IMPLICIT INTEGER (I,J,K,L,M,N)
IMPLICIT REAL (A-H, O-Z)
```

These statements represent the default in the absence of any data type specifications.

Examples of IMPLICIT statements are:

```
IMPLICIT DOUBLE PRECISION (D)
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
IMPLICIT CHARACTER*32 (T-V)
IMPLICIT CHARACTER*2 (W)
```

Type Declaration

5.2 TYPE DECLARATION STATEMENTS

Type declaration statements explicitly define the data type of specified symbolic names. There are two forms of type declaration statements: numeric type declarations (see Section 5.2.1) and character type declarations (see Section 5.2.2).

You can initialize data in either form of type declaration statement simply by placing values bounded by slashes immediately after the symbolic names of the variables or arrays to be initialized; this procedure parallels the way in which initial values are assigned in DATA statements.

The following rules apply to type declaration statements:

- Type declaration statements must precede all executable statements.
- The data type of a symbolic name can be declared only once.

5.2.1 Numeric Type Declaration Statements

Numeric type declaration statements have the form

```
type v[/clist/][,v[/clist/]]...
```

where:

type
is any data type specifier except CHARACTER.

v
is the symbolic name of a constant, variable, array, statement function or function subprogram, or array declarator.

clist
is a list of constants, as in a DATA statements. (See Section 5.9.)

You can use a numeric data type declaration statement to define arrays by including array declarators (see Section 2.5.1) in the list.

A symbolic name can be followed by a data type length specifier of the form *s, where s is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If you specify both a data type length specifier and an array declarator, the data type length specifier goes first.

You can assign initial values to variables or arrays with /clist/. /clist/ initializes only the variable or array immediately preceding it, and consists of more than one element only when used to initialize the individual elements of an array.

Examples of numeric type declaration statements are:

```
INTEGER COUNT, MATRIX(4,4), SUM
REAL MAN, MU
LOGICAL SWITCH

INTEGER*2 I, J, K, M12*4, Q, IVEC*4(10)
REAL*8 WX1, WXZ, WX3*4, WX5, WX6*8
REAL*16 P4/3.14159Q0/, E/2.72Q0/,QARRAY(10)/5*0.0, 5*1.0/
```

5.2.2 Character Type Declaration Statements

Character type declaration statements have the form

```
CHARACTER[*len] v[*len][/clist/][,v[*len][/clist/]]...
```

where:

v
is the symbolic name of a constant, variable, array, function subprogram or array declarator.

len
is an unsigned integer constant, an integer constant expression enclosed in parentheses, or an asterisk enclosed in parentheses. The value of len specifies the length of the character data elements.

clist
is a list of constants, as in a DATA statement. (See Section 5.9.)

If you specify CHARACTER*len, len is the default length specification for that list. If an item in that list does not have a length specification, the item's length is len. But if an item does have a length specification, it overrides the default length specified in CHARACTER*len.

A length specification of asterisk (for example, CHARACTER*(*)) specifies that a dummy argument or function name assumes the length specification of the corresponding actual argument or function reference (see Chapter 6). A length specification of asterisk for the symbolic name of a constant specifies that the symbolic constant assumes the actual length of the constant that it represents.

If you do not specify a length, a length of 1 is assumed. The length specification must be in the range 1 to 32767. Note that a length specification of zero is invalid. You can use a character type declaration statement to define arrays by including array declarators (see Section 2.5.1) in the list. If you specify both an array declarator and a length, the array declarator goes first.

You can assign initial values to variables or arrays with /clist/. /clist/ initializes only the variable or array immediately preceding it, and consists of more than one element only when used to initialize the individual elements in an array.

Examples of character type declaration statements follow:

```
CHARACTER*32 NAMES(100), SOCSEC(100)*9, NAMETY*10/'ABCDEFGHIJ'/
```

This statement specifies an array NAMES comprising one hundred 32-character elements, an array SOCSEC comprising one hundred 9-character elements, and a variable NAMETY, which is 10 characters long with an initial value of ABCDEFGHIJ.

```
PARAMETER (LENGTH=4)
CHARACTER*(4+LENGTH) LAST, FIRST
```

This statement specifies two 8-character variables, LAST and FIRST. (The PARAMETER statement is described in Section 5.10.)

```
SUBROUTINE S1(BUBBLE)
CHARACTER LETTER(26), BUBBLE*(*)
```

This statement specifies an array LETTER comprising twenty-six 1-character elements and a dummy argument, BUBBLE, which has a passed length (it is defined by the calling program).

```
CHARACTER*16 BIGCHR*(30000*2),QUEST*(5*INT(A))
```

This statement is invalid; the value specified for BIGCHR is too large and the length specifier for QUEST is not an integer constant expression.

DIMENSION

5.3 DIMENSION STATEMENT

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

The DIMENSION statement has the form

```
DIMENSION a(d)[,a(d)]...
```

where:

a(d)
is an array declarator. (See Section 2.5.1.)

a
is the symbolic name of an array.

d
is a dimension declarator.

The DIMENSION statement allocates a number of storage elements to each array named in the statement. One storage element is assigned to each array element in each dimension, and the length of each storage element is determined by the data type of the array. The total number of storage elements assigned to an array is equal to the product of all dimension declarators in the array declarator for that array. For example:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

This statement defines ARRAY as having 16 real elements of 4 bytes each, and defines MATRIX as having 125 integer elements of 4 bytes each.

The VIRTUAL statement has the same form and effect as the DIMENSION statement. It is provided for compatibility with PDP-11 FORTRAN.

For further information on arrays and the storage of array elements, see Section 2.5.

You can also use array declarators in type declaration and COMMON statements. However, in each program unit, you can use an array name in only one array declarator.

Examples of DIMENSION statements are:

```
DIMENSION BUD(12,24,10)
DIMENSION X(5,5,5),Y(4,85),Z(100)
DIMENSION MARK(4,4,4,4)

SUBROUTINE APROC (A1,A2,N1,N2,N3)
DIMENSION A1(N1:N2),A2(N3:*)
```

COMMON

5.4 COMMON STATEMENT

A COMMON statement defines one or more contiguous areas (blocks) of storage. A symbolic name identifies each block; however, you can omit a symbolic name for one block in a program unit. This block is the blank common block. COMMON statements also define the storage order of variables and arrays in each common block.

The COMMON statement has the form

```
COMMON [/[cb]/] nlist[[,]/[cb]/ nlist]...
```

where:

cb
is a symbolic name, called a common block name. cb can be blank. If the first cb is blank, you can omit the first pair of slashes.

nlist
is a list of variable names, array names, and array declarators separated by commas.

A common block name can be the same as a variable or array name. However, it cannot have the same name as a function, subroutine, or entry in the executable program.

When you declare common blocks of the same name in different program units, these blocks all share the same storage area when the program units are combined into an executable program.

You can have only one blank common block in an executable program, but you can have several named common blocks.

The entities in nlist must be either all of numeric data type or all of character data type. A common block cannot contain both numeric and character data.

Entities are assigned storage in common blocks on a one-for-one basis. Thus, the entities assigned by a COMMON statement in one program unit should agree in data type with entities placed in a common block by another program unit. For example, one program unit contains the statement

```
COMMON CENTS
```

and another program unit contains the statements

```
INTEGER*2 MONEY
COMMON MONEY
```

When these program units are combined into an executable program, incorrect results may occur because the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

An example of the COMMON statement follows:

Main Program

```
COMMON HEAT,X/BLK1/KILO,Q
  .
  .
CALL FIGURE
  .
  .
  .
```

Subprogram

```
SUBROUTINE FIGURE
COMMON /BLK1/LIMA,R/ /ALFA,BET
  .
  .
  .
RETURN
END
```

The COMMON statement in the main program puts HEAT and X in the blank common block, and KILO and Q in a named common block, BLK1. The COMMON statement in the subroutine makes ALFA and BET correspond to HEAT and X in the blank common block, and makes LIMA and R correspond to KILO and Q in BLK1.

You can use array declarators in the COMMON statement to define arrays.

EQUIVALENCE

5.5 EQUIVALENCE STATEMENT

The EQUIVALENCE statement partially or totally associates two or more entities in the same program unit with the same storage location.

The EQUIVALENCE statement has the form

```
EQUIVALENCE (nlist) [,(nlist)]...
```

where:

nlist
is a list of variables, array elements, arrays, or character substring references, separated by commas. You must specify at least two of these entities in each list.

The EQUIVALENCE statement allocates all of the entities in one parenthesized list beginning at the same storage location.

In an EQUIVALENCE statement, each expression in a subscript or a substring reference must be an integer constant expression.

The entities in nlist must be either of numeric data type or of character data type. You cannot make numeric entities and character entities equivalent.

You can equivalence variables of different numeric data types. If you do, multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

Examples of EQUIVALENCE statements follow.

```
DOUBLE PRECISION DVAR
INTEGER*2  IARR(4)
EQUIVALENCE (DVAR,IARR(1))
```

This EQUIVALENCE statement makes the four elements of the integer array IARR occupy the same storage as that of the double-precision variable DVAR.

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE (KEY,STAR)
```

This EQUIVALENCE statement makes the first character of the character variables KEY and STAR share the same storage location. The character variable STAR is equivalent to the substring KEY (1:10).

5.5.1 Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage space. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

You must not attempt to use the EQUIVALENCE statement to assign the same storage location to two or more elements of the same array. You also must not attempt to assign memory locations in a way that is inconsistent with the normal linear storage of array elements. For example, you cannot make the first element of one array equivalent to the first element of another array, and then attempt to set an equivalence between the second element of the first array and the sixth element of the other array.

For example:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(2,2), TRIPLE(1,2,2))
```

As a result of these statements, the entire array TABLE shares part of the storage space allocated to array TRIPLE. Figure 5-1 shows how these statements align the arrays.

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Figure 5-1 Equivalence of Array Storage

Each of the following statements also aligns the two arrays as shown in Figure 5-1:

```
EQUIVALENCE (TABLE,TRIPLE(2,2,1))
EQUIVALENCE (TRIPLE(1,1,2), TABLE(2,1))
```

Similarly, you can make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the statement

```
EQUIVALENCE (A(3,4), B(2,4))
```

The entire array A shares part of the storage space allocated to array B. Figure 5-2 shows how these statements align the arrays.

Array B		Array A	
Array Element	Element Number	Array Element	Element Number
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Figure 5-2 Equivalence of Arrays with Nonunity Lower Bounds

Each of the following statements also aligns the arrays as shown in Figure 5-2:

```
EQUIVALENCE (A,B(4,1))
EQUIVALENCE (B(3,2), A(2,2))
```

In the EQUIVALENCE statement only, you can identify an array element with a single subscript (that is, the linear element number), even though the array was defined as a multidimensional array. For example, the following statements align the two arrays as shown in Figure 5-1:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

5.5.2 **Making Substrings Equivalent**

When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets equivalences between the other corresponding characters in the character entities.

For example:

```
CHARACTER NAME*16, ID*9
EQUIVALENCE (NAME(10:13), ID(2:5))
```

As a result of these statements, the character variables NAME and ID share space as illustrated in Figure 5-3.

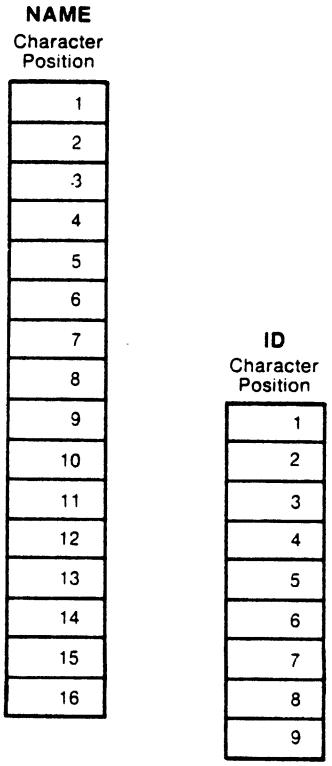


Figure 5-3 Equivalence of Substrings

The following statement also aligns the arrays as shown in Figure 5-3:

```
EQUIVALENCE (NAME(9:9),ID(1:1))
```

If the character substring references are array elements, the EQUIVALENCE statement sets equivalences between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position. For example:

```
CHARACTER FIELDS(100)*4, STAR(5)*5
EQUIVALENCE (FIELDS(1)(2:4), STAR(2)(3:5))
```

As a result of these statements, the character arrays FIELDS and STAR share storage space as shown in Figure 5-4.

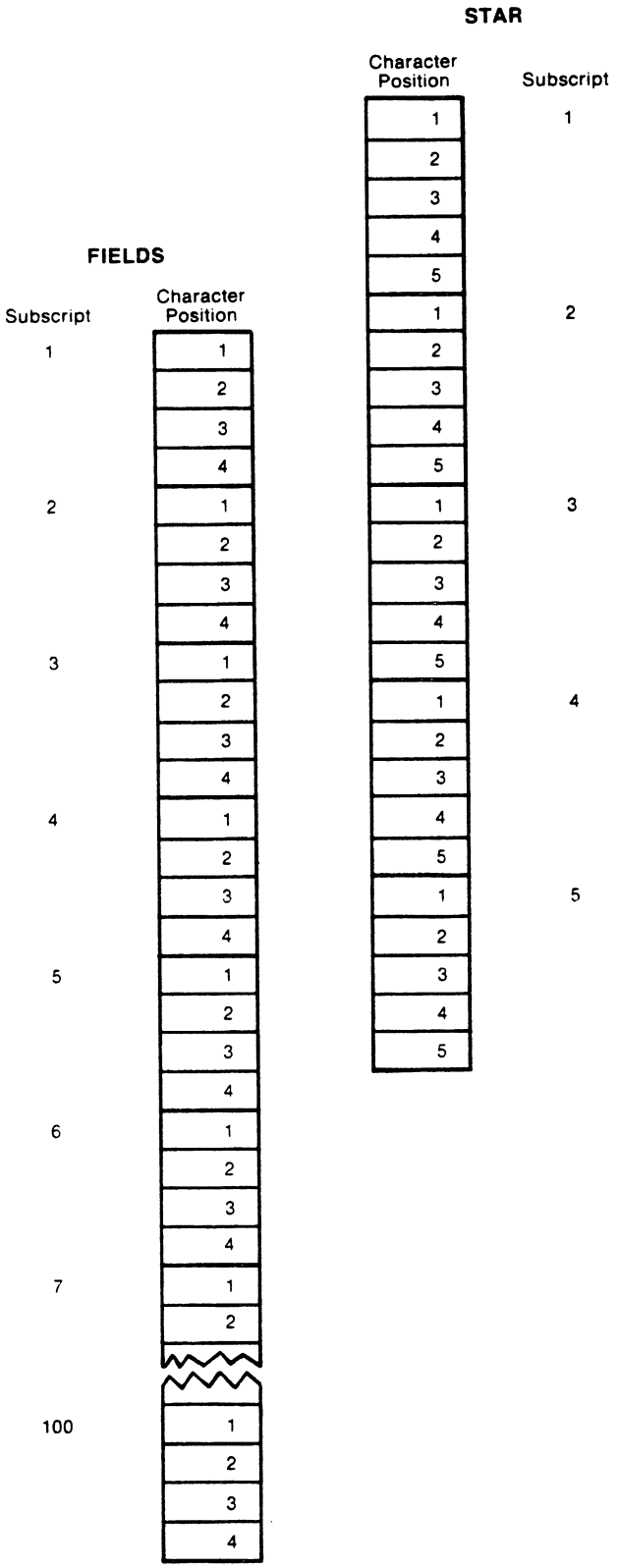


Figure 5-4 Equivalence of Character Arrays

You cannot use the EQUIVALENCE statement to assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array.

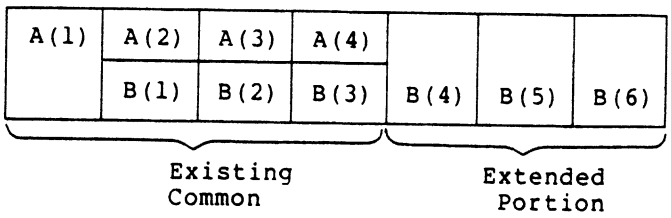
You also cannot use the EQUIVALENCE statement to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

5.5.3 EQUIVALENCE and COMMON Interaction

When you make components equivalent to entities stored in a common block, the common block can be extended beyond its original boundaries. But it can only extend beyond the last element of the previously established common block. You cannot extend the common block in such a way as to place the extended portion before the first element of the existing common block. The following examples show valid and invalid extensions of the common block:

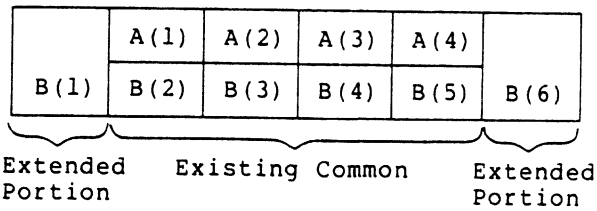
Valid

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(1))
```



Invalid

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(3))
```



If you assign two components to common blocks, you cannot make them equivalent to each other. Note also that character data can be equivalenced only with character data.

SAVE

5.6 SAVE STATEMENT

The SAVE statement retains the definition status of an entity after execution of a RETURN or END statement in a subprogram.

The SAVE statement has the form

```
SAVE [a[,a]...]
```

where:

a
is one of the following entities: a named common block name (preceded and followed by a slash), a variable name, or an array name.

An entity specified by a SAVE statement within a program unit does not become undefined upon execution of a RETURN or END statement in that unit. If the entity is in a common block, however, it may become undefined (or redefined) in another program unit.

Procedure names, the names of entities in a common block, and dummy argument names cannot be used in a SAVE statement.

A SAVE statement that does not explicitly contain a list is treated as though it contained a list consisting of all allowable items in the program unit in which the SAVE statement resides.

EXTERNAL

5.7 EXTERNAL STATEMENT

The EXTERNAL statement allows you to use external procedure names as arguments to other subprograms.

The subprograms to be used as arguments can never be FORTRAN intrinsic functions; they can only be user-supplied functions and subroutines. The INTRINSIC statement discussed in Section 5.8 allows intrinsic function names to be used as arguments.

The EXTERNAL statement has the form

```
EXTERNAL v[,v]...
```

where:

v
is the symbolic name of a user-supplied subprogram, or the name of a dummy argument associated with the name of a subprogram.

The EXTERNAL statement declares each symbolic name included in it to be the name of an external procedure. This name can then be used as an actual argument to a subprogram which can use the corresponding dummy argument in a function reference or a CALL statement.

Note that a complete function reference used as an argument -- FUNC(B) in CALL SUBR (A, FUNC(B), C), for example -- represents a value, not a subprogram. A complete function reference is not, therefore, defined in an EXTERNAL statement.

The interpretation of the EXTERNAL statement described above is different from that of earlier versions of Digital FORTRAN. See Appendix A for the earlier interpretation.

INTRINSIC

5.8 INTRINSIC STATEMENT

The INTRINSIC statement allows you to use intrinsic function names as arguments to subprograms. Sections C-3 and C-4 contain the names and descriptions of the individual FORTRAN intrinsic functions; for further information on intrinsic functions, see Chapter 6.

The INTRINSIC statement has the form

```
INTRINSIC v[,v]...
```

where:

v
is the symbolic name of an intrinsic function.

The INTRINSIC statement declares each symbolic name included in it to be the name of an intrinsic procedure. This name can then be used as an actual argument to a subprogram which can use the corresponding dummy argument in a function reference or a CALL statement.

An example of the use of the INTRINSIC statement follows:

```
          Main Program

      .
      .
      .
EXTERNAL CTN
INTRINSIC SIN, COS
      .
      .
      .
CALL TRIG (ANGLE, SIN, SINE)
      .
      .
      .
CALL TRIG (ANGLE, COS, COSINE)
      .
      .
      .
CALL TRIG (ANGLE, CTN, COTANGENT)
      .
      .
      .

          Subprograms

SUBROUTINE TRIG (X,F,Y)
Y=F(X)
RETURN
END

FUNCTION CTN(X)
CTN=COS(X)/SIN(X)
RETURN
END
```

In this example, when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the FORTRAN library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

DATA

5.9 DATA STATEMENT

The DATA statement assigns initial values to variables and array elements before program execution.

The DATA statement has the form

```
DATA nlist/clist/[[,]nlist/clist/]...
```

where:

nlist
is a list of one or more variable names, array names, array element names, character substring names, or implied-DO lists, separated by commas. Subscript expressions and expressions in substring references must be integer expressions containing integer constants and implied-DO variables. The form of an implied DO list in a DATA statement is:

```
(dlist,i=n1,n2[,n3])
```

where:

dlist
is a list of one or more array element names, character substring names, or implied-DO lists, separated by commas.

i
is the name of an integer variable.

n1,n2,n3
are integer expressions that may contain integer constants and implied-DO variables.

clist
is a list of constants; clist constants have one of the following forms:

```
value

n * value
```

where:

n
is used when clist is specified as n * value. It defines the number of times the same value is to be assigned to successive entities in the associated nlist; n is a nonzero, unsigned integer constant or the symbolic name of an integer constant.

The DATA statement assigns the constant values in each clist to the entities in the preceding nlist. Values are assigned one by one in order as they appear, from left to right.

The number of constants must correspond exactly to the number of entities in the preceding nlist.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.

If both the constant value in the clist and the entity in the nlist have numeric data types, the conversion is based on the following rules:

- The constant value is converted, if necessary, to the data type of the variable being initialized.
- When an octal or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the component. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeroes; if the constant contains more digits than can be stored, the constant is truncated on the left.
- When a Hollerith or character constant is assigned to a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the component (see Table 2-2). If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with spaces. If the constant contains more characters than can be stored, the constant is truncated on the right.

If the constant value in the clist and the entity in the nlist are both character data type, the conversion is based on the following rules:

- If the constant contains fewer bytes than the length of the entity, the rightmost character positions of the entity are initialized with spaces.
- If the constant contains more bytes than the length of the entity, the character constant is truncated on the right.

If the constant value is numeric data type and the entity in the nlist is character data type, the constant and the entity must conform to these restrictions:

- The character entity must have a length of one character.
- The constant must be an integer, octal, or hexadecimal constant and must have a value in the range 0 through 255.

When the constant and the entity conform to these restrictions, the entity is initialized with the character that has the ASCII code specified by the constant. This permits a character entity to be initialized to any 8-bit ASCII code.

An example of the DATA statement follows:

```
INTEGER A(10)
CHARACTER BELL,TAB,LF,FF, STARS*6
DATA A,STARS / 10*0, '****' /
DATA BELL,TAB,LF,FF /7,9,10,12/
```

The DATA statements assign zero to all 10 elements of array A, and 4 asterisks followed by 2 spaces to the character variable STARS. ASCII control character codes are assigned to the character variables BELL, TAB, LF, and FF.

PARAMETER

5.10 PARAMETER STATEMENT

The PARAMETER statement assigns a symbolic name to a constant.

The PARAMETER statement has the form

```
PARAMETER (p=c[,p=c]...)
```

where:

p
is a symbolic name.

c
is a constant, the symbolic name of a constant, or a compile-time constant expression.

Each symbolic name (p) becomes a constant and is defined as the value (c) to which it is equated, where c is any valid FORTRAN constant.

A compile-time constant expression can be a compile-time logical expression, a compile-time character expression, or a compile-time arithmetic expression.

A compile-time logical expression is a logical expression in which:

- Each operand is a constant, the symbolic name of a constant, or another compile-time constant expression.
- Each operand is of the logical or integer data type.
- Each operator is a Boolean or relational operator.

A compile-time character expression is a character expression in which:

- Each operand is a constant, the symbolic name of a constant, or another compile-time constant expression.
- Each operand is of the character data type.
- Each operator is the concatenation operator //.

A compile-time arithmetic expression is an arithmetic expression in which:

- Each operand is a constant, the symbolic name of a constant, or another compile-time constant expression.
- Each operand is of integer, real, or complex data type.
- Each operator is a +, -, *, /, or ** operator. (The ** operator is valid only if the exponent is of the integer data type.)

The data type of a symbolic name defined to be a constant is determined by the same rules for implicit declarations that determine the data type of any other symbolic name, or by an explicit type declaration statement preceding the defining PARAMETER statement. Therefore, MU=1.23 in a PARAMETER statement will be interpreted as MU=1, unless the PARAMETER statement is preceded by an appropriate type declaration or IMPLICIT statement (for example, REAL*8 MU).

Once a symbolic name is defined to be a constant, it can appear any place in a program that any other constant can appear. The effect is that of writing the value that corresponds to the symbolic name instead of the name itself.

The symbolic name of a constant cannot appear as part of another constant, though it can appear as either the real or imaginary part of a complex constant.

You can use a symbolic name defined to be a constant only within the program unit containing the defining PARAMETER statement. Also, a symbolic name can be defined only once within the same program unit.

The form and the interpretation of the PARAMETER statement described above are different from those of the PARAMETER statement provided in earlier versions of Digital FORTRAN. However, VAX-11 FORTRAN provides both the FORTRAN-77 and the earlier form of the PARAMETER statement; see Appendix A for information on the earlier form and interpretation.

The following sequence demonstrates the use of the FORTRAN-77 PARAMETER statement:

```
REAL*4 PI,PIOV2
REAL*8 DPI,DPIOV2
LOGICAL FLAG
CHARACTER*(*) LONGNAME
PARAMETER (PI=3.1415927, DPI=3.14159265358979238D0)
PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)
PARAMETER (FLAG=.TRUE.,LONGNAME='A STRING OF 25 CHARACTERS')
```

PROGRAM

5.11 PROGRAM STATEMENT

The PROGRAM statement assigns a symbolic name to a main program unit.

The PROGRAM statement has the form

```
PROGRAM nam
```

where:

nam
is a symbolic name.

The PROGRAM statement is optional. If you use it, it must be the first statement in the main program. The symbolic name must not be the name of any entity within the main program, and also must not be the same as the name of any subprogram, entry, or common block in the same executable program.

BLOCK DATA

5.12 BLOCK DATA STATEMENT

The BLOCK DATA statement, followed by a series of specification statements, assigns initial values to entities in named common blocks and, at the same time, establishes and defines these blocks.

The BLOCK DATA statement has the form

```
BLOCK DATA [nam]
```

where:

nam
is a symbolic name.

You can use type declaration, IMPLICIT, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, SAVE, and DATA statements following a BLOCK DATA statement.

The specification statements that follow the BLOCK DATA statement establish and define common blocks, assign variables and arrays to these blocks, and assign initial values to the variables and arrays.

A BLOCK DATA statement and its associated specification statements comprise a special kind of program unit. The last statement in a BLOCK DATA program unit is an END statement.

A BLOCK DATA program unit must not contain any executable statements.

If you use a BLOCK DATA statement to initialize any entity in a labeled common block, you must provide a complete set of specification statements to establish the entire block, even though some of the entities in the block do not appear in a DATA statement. You can use the same BLOCK DATA program unit to define initial values for more than one common block.

An example of a BLOCK DATA program unit follows:

```
BLOCK DATA BLKDAT
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/AREA2/W,X,Y
DATA R/1.0,2*2.0/,T/.FALSE./,U/0.214537D-7/,W/.TRUE./,Y/3.5/
END
```

CHAPTER 6

SUBPROGRAMS

Subprograms are program units that can be invoked from another program, usually to perform some commonly used computation on behalf of the other program.

Subprograms are either user-supplied, or supplied as part of the VAX-11 FORTRAN system. User-supplied subprograms include:

- Statement functions
- Function subprograms
- Subroutines

Subprograms supplied with the FORTRAN system include:

- Intrinsic mathematical functions
- Intrinsic character functions
- Miscellaneous intrinsic functions

Normally, the program invoking the subprogram passes values -- known as actual arguments -- to the subprogram, which uses the actual arguments to compute the results.

6.1 SUBPROGRAM ARGUMENTS

Subprogram arguments are either dummy arguments or actual arguments. Dummy arguments are specified when you define the subprogram. Actual arguments are specified when you invoke the subprogram, and are associated with the corresponding dummy arguments when control is transferred to the subprogram. This means that each dummy argument takes on the value of the corresponding actual argument; and that any value assigned to the dummy argument is also assigned to the corresponding actual argument.

6.1.1 Actual Argument And Dummy Argument Association

Actual arguments must agree in order, number and data type with the dummy arguments with which they are associated. Actual arguments can be constants, variables, expressions, arrays, array elements, character substrings, alternate return specifiers, or subprogram names. The dummy arguments specified in subprogram definitions, representing corresponding actual arguments, appear as unsubscripted variable names.

Although dummy arguments are not actual variables, arrays, or subprograms, each dummy argument may be declared as though it were a variable, array, or subprogram.

A dummy argument declared as an array can be associated only with an actual argument that is an array or array element of the same data type. If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays (see Section 6.1.1.1) to process arrays of different sizes in a single subprogram.

The length of a dummy argument of type character must not be greater than the length of its associated actual argument. Note that if the character dummy argument's length is specified as *(*), the length used is exactly the length of the associated actual argument. (This is known as a passed length character argument. See Section 6.1.1.3.)

Section 6.1.1.4 describes the use of character constants and Hollerith constants as actual arguments.

Section 6.1.1.5 describes alternate return arguments.

6.1.1.1 **Adjustable Arrays** - Adjustable arrays are dummy arguments in subprograms. The dimensions of an adjustable array are determined in the reference to the subprogram. The array declarator (see Section 2.5.1) for an adjustable array can contain integer variables that are dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument used in the array declarator must be associated with an actual argument and each variable in a common block used in an array declarator must have a defined value. The dimension declarator is evaluated using the values of the actual arguments, variables in common blocks, and constants specified in the array declarator.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the dummy arguments M and N to control the iteration.

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,N)
      SUM = 0.0
      DO 10 J = 1,N
      DO 10 I = 1,M
   10 SUM = SUM + A(I,J)
      RETURN
      END
```

The following statements are sample calls on SUM:

```
DIMENSION A1(10,35), A2(3,56)
SUM1 = SUM(A1,10,35)
SUM2 = SUM(A2,3,56)
SUM3 = SUM(A1,10,10)
```

The upper- and lower-dimension bound values are determined once each time a subprogram is entered. These values do not change during the execution of that subprogram even if the values of variables contained in the array declaration are changed. For example:

```
DIMENSION ARRAY (11,5)
L = 9
M = 5
CALL SUB(ARRAY,L,M)
END

SUBROUTINE SUB(X,I,J)
DIMENSION X(-I/2:I/2,J)
X(I/2,J)= 999
J = 1
I = 2
END
```

In this example, the adjustable array X is declared as X(-4:4,5) on entry to subroutine SUB. The assignments to I and J do not affect that declaration.

A variable used in a bounds expression of an adjustable array must not, after that variable has appeared in an array declaration, also appear in a type declaration that changes the variable's data type. The following program segment is invalid:

```
SUBROUTINE SUB1(A,X)
DIMENSION A(X)
INTEGER X
```

An adjustable array is undefined if a dummy argument array is not currently associated with an actual argument array, or if any of the variables in the adjustable array declarator are either not currently associated with an actual argument or are not in a common block. Note that argument association is not retained between one reference to a subprogram and the next reference to that subprogram. For example:

```
SUBROUTINE S(A,I,X)
DIMENSION A(I)
A(I) = X
RETURN
ENTRY S1 (I,A,K,L)
A(I) = A(I) + 1.0
RETURN
END
```

In this example, B is a real array with 10 elements:

```
DIMENSION B(10)
```

The following statement sets B(2)=3.0:

```
CALL S(B,2,3.0)
```

The next statement increments B(5) by 1.0:

```
CALL S1(5,B,3,2)
```

6.1.1.2 **Assumed-Size Dummy Arrays** - An assumed-size dummy array is a dummy array for which the upper bound of the last dimension is specified as *. For example:

```
SUBROUTINE SUB(A,N)
DIMENSION A(1:N,1:*)
.
.
.
```

The size of an assumed-size array, and the number of elements that can be referenced, are determined as follows:

- If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual-argument array.
- If the actual argument corresponding to the dummy argument is a noncharacter array element name, with a subscript value of s in an array of size a, the size of the dummy array is a+1-s.
- If the actual argument is a character-array name, character-array element name, or character-array element substring name, and begins at character storage unit b of an array with n character storage units, the size of the dummy array is INT(n+1-b)/y, where y is the length of an element of the dummy array.

Because the actual size of an assumed-size array is not known, an assumed-size array name cannot be used as:

- An array name in the list of an I/O statement
- A unit identifier for an internal file in an I/O statement
- A run-time format specifier in an I/O statement.

6.1.1.3 **Passed Length Character Arguments** - A passed length character argument must be a dummy argument, though it has the length of the actual argument.

For a passed length character string, use an asterisk enclosed in parentheses as the length specification (see Section 6.2.2.2).

When control transfers to a subprogram, each passed length character dummy argument must be associated with a character actual argument. The length of the dummy argument is the length of the actual argument.

A character array dummy argument can have passed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The passed length and the array declarator together determine the size of the passed length character array. A passed length character array can also be an adjustable or assumed-size array.

The following example of a function subprogram uses a passed length character argument. The function finds the position of the character with the highest ASCII code value; it uses the length of the passed length character argument to control the iteration. (Note that the processor-defined function LEN is used to determine the length of the argument. See Section 6.3.4 for a description of the LEN function.)

```
      INTEGER FUNCTION ICMAX(CVAR)
      CHARACTER*(*) CVAR
      ICMAX = 1
      DO 10 I = 2, LEN(CVAR)
   10 IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
      RETURN
      END
```

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
   .
   .
   .

I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX (VAR(3:4)//CARRAY(3,5))
```

6.1.1.4 **Character and Hollerith Constants as Actual Arguments** -Actual arguments and their corresponding dummy arguments must agree in data type. If the actual argument is a Hollerith constant (for example, 4HABCD), the dummy argument must be of numeric data type. In VAX-11 FORTRAN, if an actual argument is a character constant (for example, 'ABCD'), the corresponding dummy argument can have either numeric or character data type. If the dummy argument has a numeric data type, the character constant 'ABCD' is, in effect, converted to a Hollerith constant by the FORTRAN compiler and the linker.

An exception to this occurs when the function or subroutine name is itself a dummy argument. It is not possible to determine at compile time or link time whether a character constant or Hollerith constant is required. In this case, a character constant actual argument can correspond only to a character dummy argument. For example:

```
SUBROUTINE S(CHARSUB,HOLLSUB,A,B)
EXTERNAL CHARSUB,HOLLSUB
   .
   .
   .

CALL CHARSUB(A,'STRING')
CALL HOLLSUB(B,6HSTRING)
```

In this example, the subroutine names CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument 'STRING' in the call to CHARSUB must correspond to a character dummy argument, whereas the actual argument 6HSTRING in the call to HOLLSUB must correspond to a Hollerith dummy argument.

6.1.1.5 **Alternate Return Arguments** - To specify an alternate return argument in a dummy argument list, place asterisks in the list. For example:

```
SUBROUTINE MINN(A,B,*,*,C)
```

The actual argument list passed in the CALL must include alternate return arguments in the corresponding positions. These arguments have the form

```
*label
```

or

```
&label
```

You can use either an asterisk or an ampersand to indicate an alternate return argument in an actual argument list. The value you specify for label must be the label of an executable statement in the program that issued the CALL.

6.1.2 Built-In Functions

Built-in functions perform utility operations that are useful in communicating with subprograms written in languages other than FORTRAN. The two kinds of built-in functions are:

- Argument list built-in functions
- %LOC built-in function

6.1.2.1 **Argument List Built-In Functions** - To call subprograms (such as VAX/VMS system services) written in languages other than FORTRAN, you may need to pass the actual arguments in a form different from that used by FORTRAN. You can use three built-in functions -- %VAL, %REF, and %DESCR -- in the argument list of a CALL statement or function reference to change the form of the argument.

You do not need to use these built-in functions to invoke a FORTRAN library procedure or a user-supplied subprogram written in FORTRAN.

The three argument list built-in functions specify the way the argument should be passed to the subprogram. You can use these functions only in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

The argument list built-in functions are:

Function	Effect
%VAL(a)	Pass the argument as a 32-bit immediate value
%REF(a)	Pass the argument by reference
%DESCR(a)	Pass the argument by descriptor

In these functions, a is an actual argument.

See the VAX-11 FORTRAN User's Guide for more information on argument-passing mechanisms.

Table 6-1 lists the FORTRAN argument-passing defaults and the allowed uses of %VAL, %REF, and %DESCR.

Table 6-1
Argument List Built-In Functions and Defaults

Actual Argument Data Type	Default	Functions Allowed		
		%VAL	%REF	%DESCR
Expressions				
Logical	REF	Yes	Yes	Yes
Integer	REF	Yes	Yes	Yes
REAL*4	REF	Yes	Yes	Yes
REAL*8	REF	No	Yes	Yes
REAL*16	REF	No	Yes	Yes
Complex	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Hollerith	REF	No	No	No
Array Name				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes
Procedure Name				
Numeric	REF	No	Yes	Yes
Character	DESCR	No	Yes	Yes

6.1.2.2 **%LOC Built-In Function** - The %LOC built-in function computes the internal address of a storage element. It has the form

```
%LOC(v)
```

where:

v
is a variable name, array element name, array name, character substring name, or external procedure name.

The %LOC built-in function produces an INTEGER*4 value that represents the location of its argument. It can be used as an element in an arithmetic expression.

See the VAX-11 FORTRAN User's Guide for more information on the %LOC built-in function.

6.2 USER-WRITTEN SUBPROGRAMS

A user-written subprogram is a FORTRAN statement or group of FORTRAN statements that performs a computing procedure. A computing procedure can be either a series of arithmetic operations or a series of FORTRAN statements. You can use subprograms to perform a computing procedure in several places in your program, and thus avoid duplicating the series of operations or statements in each place.

There are three types of subprograms. Table 6-2 lists each type of subprogram, the statements needed to define it, and the method of transferring control to the subprogram.

Table 6-2
Types of User-Written Subprograms

Subprogram	Defining Statements	Control Transfer Method
Statement function	Statement function definition	Function reference
Function	FUNCTION ENTRY	Function reference
Subroutine	SUBROUTINE ENTRY	CALL statement

A function reference is used in an expression and consists of the function name and the function arguments. A function reference returns a value that is used in evaluating the expression in which it appears.

Function and subroutine subprograms can change the values of their arguments; the calling program can use the changed values.

A subprogram can refer to other subprograms; but it cannot, either directly or indirectly, refer to itself.

6.2.1 Statement Functions

A statement function is a computing procedure defined by a single statement. Its definition statement is similar in form to an assignment statement. If you refer to the statement function, the computation is performed. The resulting value is made available to the expression that contains the statement function reference.

The statement function definition statement has the form

```
f ([p[,p]...])=e
```

where:

f
is the name of the statement function.

p
is a dummy argument.

e
is an expression.

The expression (e) is an arithmetic, logical, or character expression that defines the computation to be performed.

A statement function reference has the form

```
f ([p[,p]...])
```

where:

f
is the name of the function.

p
is an actual argument.

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of a statement function is determined either implicitly by the initial letter of the function name, or explicitly in a type declaration statement. The data type can be any of the data types, including the character data type.

Dummy arguments in a statement function indicate only the number, order, and data type of the actual arguments. You can use the names of the dummy arguments to represent other entities elsewhere in the program unit. Note that, except for data type, declarative information associated with an entity is not associated with the dummy arguments in the statement function. That is, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.

You cannot use the name of the statement function to represent any other entity within the same program unit.

The expression in a statement function definition can contain function references. If a reference to another statement function appears in the expression, you must have previously defined that function in the same program unit.

Any reference to a statement function must appear in the same program unit as the definition of that function.

A statement function reference must appear as, or be part of, an expression. You cannot use the reference as the left side of an assignment statement.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

Examples of statement function definitions follow:

```
VOLUME(RADIUS) = 4.189*RADIUS**3

SINH(X) = (EXP(X)-EXP(-X))*0.5

CHARACTER*10 CSF,A,B
CSF(A,B) = A(6:10)//B(1:5)
```

The following is an invalid definition. You cannot use a constant as a dummy argument.

```
AVG(A,B,C,3.) = (A+B+C)/3.
```

Examples of statement function references follow:

This is the definition:

```
AVG(A,B,C) = (A+B+C)/3.
```

These are the references:

```
      .
      .
      .
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q).LT.AVG(X,Y,Z)) GO TO 300
FINAL = AVG(TEST3,TEST4,LAB2)
```

The last of these three references is invalid because the data type of the third argument does not agree with the dummy argument.

6.2.2 Function Subprograms

A function subprogram is a program unit consisting of a FUNCTION statement followed by a series of statements that define a computing procedure. You use a function reference to transfer control to a function subprogram, and a RETURN or END statement to return control to the calling program unit.

A function subprogram returns a single value to the calling program unit by assigning that value to the function's name. The function's name determines the data type of the value returned.

6.2.2.1 Logical and Numeric Functions

The FUNCTION statement has the form

```
[typ] FUNCTION nam[*m][([p[,p]...])]
```

where:

typ
is one of the logical or numeric data type specifiers.

nam
is the name of the function.

m
is an unsigned, nonzero integer constant specifying the length of the data type; it must be one of the valid length specifiers for the data type given by typ.

p
is a dummy argument.

6.2.2.2 **Character Functions** - The CHARACTER FUNCTION statement has the form

```
CHARACTER[*n] FUNCTION nam[*n] [([p[,p]...])]
```

where:

n
is an unsigned, nonzero integer constant, or (*) indicating a passed length function name. If you specify CHARACTER*(*), the function assumes the length declared for it in the program unit that invokes it. A passed length character function can have different lengths when it is invoked by different program units. If n is an integer constant, the value of n must agree with the length of the function specified in the program unit that invokes the function. If you do not specify n, a length of 1 is assumed. If the length has already been specified following the keyword CHARACTER, the optional length specification following nam is not permitted.

nam
is the name of the function.

p
is a dummy argument.

6.2.2.3 **Function Reference** - A function reference that transfers control to a function subprogram has the form

```
nam ([p[,p]...])
```

where:

nam
is the symbolic name of the function.

p
is an actual argument.

When control transfers to a function subprogram, the values of the actual arguments (if any) in the function reference are associated with the dummy arguments (if any) in the FUNCTION statement. The statements in the subprogram are then executed. A value must be assigned to the name of the function. Finally, a RETURN statement is executed in that function and returns control to the calling program unit. The value assigned to the function's name is now available to the expression containing the function reference, and is used to complete the evaluation of that expression.

The data type of a function name can be specified explicitly in the FUNCTION statement or in a type declaration statement, or it can be specified implicitly. The function name defined in the function subprogram must have the same data type as the function name in the calling program unit.

The FUNCTION statement must be the first statement of a function subprogram. A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA statement, a PROGRAM statement, or another FUNCTION statement. ENTRY statements can be included to provide multiple entry points to the subprogram (see Section 6.2.4).

Examples of function subprograms follow:

```
      FUNCTION ROOT(A)
      X = 1.0
2     EX = EXP(X)
      EMINX = 1./EX
      ROOT = ((EX+EMINX)*.5+COS(X)-A)/((EX - EMINX)*.5-SIN(X))
      IF (ABS(X-ROOT).LT.1E-6) RETURN
      X = ROOT
      GO TO 2
      END
```

The function in this example uses the Newton-Raphson iteration method to obtain the root of the function:

$$F(X) = \cosh(X) + \cos(X) - A = 0$$

The value of A is passed as an argument. The iteration formula for this root is:

$$X_{i+1} = X_i - \left(\frac{\cosh(X_i)+\cos(X_i)-A}{\sinh(X_i)-\sin(X_i)} \right)$$

This is calculated repeatedly until the difference between Xi and Xi+1 is less than 1.0E-6. The function uses the FORTRAN intrinsic functions EXP, SIN, COS, and ABS (see Section 6.3).

The following example is a passed length character function. It returns the value of its argument, repeated to fill the length of the function.

```
      CHARACTER*(*) FUNCTION REPEAT(CARG)
      CHARACTER*1 CARG
      DO 10 I=1, LEN(REPEAT)
10    REPEAT(I:I) = CARG
      RETURN
      END
```

Within any given program unit all references to a passed length character function must have the same length. In the following example, the REPEAT function has a length of 1000:

```
      CHARACTER*1000 REPEAT, MANYAS, MANYZS
      MANYAS = REPEAT('A')
      MANYZS = REPEAT('Z')
```

However, another program unit within the executable program can specify a different length. In the following example, the REPEAT function has a length of 2:

```
      CHARACTER HOLD*6, REPEAT*2
      HOLD = REPEAT('A') // REPEAT('B') // REPEAT('C')
```

6.2.3 Subroutine Subprograms

A subroutine subprogram is a program unit consisting of a SUBROUTINE statement followed by a series of statements that define a computing procedure. You use a CALL statement to transfer control to a subroutine subprogram, and a RETURN or END statement to return control to the calling program unit.

The SUBROUTINE statement has the form

```
SUBROUTINE nam [([p[,p]...])]
```

where:

nam
is the name of the subroutine.

p
is a dummy argument. You can specify a dummy argument as an alternate return argument by placing an asterisk in the argument list.

Section 4.6 describes the CALL statement.

When control transfers to the subroutine, the values of the actual arguments (if any) in the CALL statement are associated with the corresponding dummy arguments (if any) in the SUBROUTINE statement. The statements in the subprogram are then executed.

The SUBROUTINE statement must be the first statement of a subroutine.

A subroutine subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, a PROGRAM statement, or another SUBROUTINE statement. ENTRY statements are allowed to specify multiple entry points in the subroutine (see Section 6.2.4).

Examples:

The subroutine in the following example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses the computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron. The GO TO statement also transfers control to the proper procedure for calculating the volume. If the number of faces is not 4, 6, 8, 12, or 20, the subroutine displays an error message on the user's terminal.

Main Program

```
COMMON NFACES,EDGE,VOLUME
ACCEPT *, NFACES,EDGE
CALL PLYVOL
TYPE *, 'VOLUME=',VOLUME
STOP
END
```

Subroutine

```
      SUBROUTINE PLYVOL
      COMMON NFACES,EDGE,VOLUME
      CUBED = EDGE**3
      GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5),NFACES
      GOTO 6
    1 VOLUME = CUBED * 0.11785
      RETURN
    2 VOLUME = CUBED
      RETURN
    3 VOLUME = CUBED * 0.47140
      RETURN
    4 VOLUME = CUBED * 7.66312
      RETURN
    5 VOLUME = CUBED * 2.18170
      RETURN
    6 TYPE 100, NFACES
  100 FORMAT(' NO REGULAR POLYHEDRON HAS ',I3,'FACES.'/)
      VOLUME=0.0
      RETURN
      END
```

The following example illustrates the use of alternate return specifiers to determine where control is transferred on completion of the subroutine. The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list. The RETURN taken depends on the value of Z, as computed in the subroutine. Thus, if Z is less than zero, the normal return is taken; if equal to zero, the return is to statement label 10 in the main program; if greater than zero, the return is to statement label 20 in the main program.

Main Program

```
      CALL CHECK (A,B,*10,*20,C)
      TYPE *, 'VALUE LESS THAN ZERO'
      GO TO 30
10    TYPE *, 'VALUE EQUALS ZERO'
      GO TO 30
20    TYPE *, 'VALUE MORE THAN ZERO'
30    CONTINUE
         .
         .
         .
```

Subroutine

```
      SUBROUTINE CHECK (X,Y,*,*,Q)
         .
         .
         .
50    IF(Z) 60,70,80
60    RETURN
70    RETURN 1
80    RETURN 2
      END
```

ENTRY

6.2.4 ENTRY Statement

The ENTRY statement provides multiple entry points within a subprogram. It is not executable and can appear within a function or subroutine program after the FUNCTION or SUBROUTINE statement. Execution of a subprogram referred to by an entry name begins with the first executable statement after the ENTRY statement.

The ENTRY statement has the form

```
ENTRY nam [([p[,p]...])]
```

where:

nam
is the entry name.

p
is a dummy argument.

Use the CALL statement to refer to entry names within subroutine subprograms. Use function references to refer to entry names within function subprograms.

An entry name within a function subprogram can appear in a type declaration statement.

You can specify an entry name in an EXTERNAL statement and use it as an actual argument; you cannot use it as a dummy argument.

You cannot use entry names in executable statements that physically precede the appearance of the entry name in an ENTRY statement.

You can include alternate return arguments in ENTRY statements by placing asterisks in the dummy argument list. ENTRY statements that specify alternate return arguments can be used only in subroutine subprograms.

You can use dummy arguments in ENTRY statements that differ in order, number, type, and name from the dummy arguments you use in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

A dummy argument can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. A dummy argument is undefined if it is not currently associated with an actual argument. An argument association is not retained from one reference of a subprogram to the next.

You cannot use an ENTRY statement within a block IF construct or a DO loop.

6.2.4.1 **ENTRY in Function Subprograms** - All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names of the same data type; all associated names that are of different data types become undefined. The function and entry names need not be of the same data type, but they all must be consistent within one of the following groups of data types:

Group 1: BYTE, INTEGER*2, INTEGER*4, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8, COMPLEX*8

Group 2: COMPLEX*16, REAL*16

Group 3: CHARACTER

At the execution of a RETURN statement or the implied return at the end of the subprogram, the name used to refer to the function subprogram must be defined.

If the function is of character data type, all entry names must also be of character data type and must have the same length specification as that of the function. Note that the length specified must also agree with the length specified in the program unit referring to the entry name. If an asterisk enclosed in parentheses is used to specify the length of the entry name, the entry name has a passed length (see Section 6.1.1.3).

Figure 6-1 illustrates a function subprogram that computes the hyperbolic functions sinh, cosh, and tanh.

```
      REAL FUNCTION TANH(X)
C
C     STATEMENT FUNCTION TO COMPUTE TWICE SINH
C
      TSINH(Y) = EXP(Y) - EXP (-Y)
C
C     STATEMENT FUNCTION TO COMPUTE TWICE COSH
C
      TCOSH(Y) = EXP(Y) + EXP(-Y)
C
C     COMPUTE TANH
C
      TANH = TSINH(X) / TCOSH(X)
      RETURN
C
C     COMPUTE SINH
C
      ENTRY SINH(X)
      SINH = TSINH(X) / 2.0
      RETURN
C
C     COMPUTE COSH
C
      ENTRY COSH(X)
      COSH = TCOSH(X) / 2.0
      RETURN
      END
```

Figure 6-1 Multiple Functions in a Function Subprogram

6.2.4.2 **ENTRY in Subroutine Subprograms** - To refer to an entry point name in a subroutine, issue a CALL statement that includes the entry point name defined in the ENTRY statement. For example:

Main Program

```
CALL SUBA(A,B,C)
   .
   .
   .
```

Subroutine

```
SUBROUTINE SUB (X,Y,Z)
   .
   .
   .
ENTRY SUBA(Q,R,S)
```

In this example, the CALL is to an entry point (SUBA) within the subroutine (SUB). Execution begins with the first statement following ENTRY SUBA (Q,R,S), using the actual arguments (A,B,C) passed in the CALL statement. Note that alternate returns can be specified in ENTRY statements. For example:

```
SUBROUTINE SUB (K,*,*)
     .
     .
     .
ENTRY SUBC (J,K,*,*,X)
     .
     .
     .
RETURN 1
RETURN 2
END
```

If you issue a CALL to entry point SUBC, you must include actual alternate return arguments. For example:

```
CALL SUBC(M,N,*100,*200,P)
```

In this case, RETURN 1 transfers control to statement label 100 and RETURN 2 transfers control to statement label 200, in the calling program.

6.3 FORTRAN INTRINSIC FUNCTIONS

FORTRAN library functions, called intrinsic functions, are provided to perform commonly used mathematical computations.

The FORTRAN intrinsic functions are listed in Appendix C. Function references to FORTRAN intrinsic functions are written in the same way function references to user-defined functions are written. For example:

```
R = 3.14159 * ABS(X-1)
```

As a result of this reference, the absolute value of X-1 is calculated and multiplied by the constant 3.14159; the result is assigned to the variable R.

Appendix C gives the data type of each intrinsic function and that of its actual arguments. Section 6.3.4 also describes the character functions.

6.3.1 Intrinsic Function References

FORTRAN library function names are called intrinsic function names. Normally, a name in the table of intrinsic function names (Table C-1) refers to the FORTRAN library function with that name. However, the name can refer to a user-defined function when the name appears in an EXTERNAL statement.

Except when they are used in an EXTERNAL statement, intrinsic function names are local to the program unit that refers to them. Thus, they can be used for other purposes in other program units. In addition, the data type of an intrinsic function does not change if you use an IMPLICIT statement to change the implied data type rules.

Note that you cannot have an intrinsic function and a user-defined function with the same name in the same program unit.

6.3.2 Generic Function References

Generic function references provide a way of calling some of the FORTRAN mathematical functions such that the selection of the actual library routine used is based on the data type of the argument in the function reference. For example, if X is a REAL variable, the function reference SIN(X) refers to the real-valued sine function. But if D is a REAL*8 variable, the function reference SIN(D) refers to the REAL*8 sine function. You need not write DSIN(D).

Generic function selection occurs independently for each function reference. Thus, you could use both the function references in the example above, SIN(X) and SIN(D), in the same program unit.

Table 6-3 lists the generic function names. You can use these names only with the argument data types shown in the table.

You cannot use the names in Table 6-3 for generic function selection if you use them in a program unit in any of the following ways:

- As the name of a statement function
- As a dummy argument name, a common block name, or a variable or array name

Using a generic name in an INTRINSIC statement does not affect generic function selection for function references. When you use a generic function name in an actual argument list as the name of a function to be passed, no generic function selection occurs, as there is no argument list on which to base a selection. The name is treated according to the rules for nongeneric FORTRAN functions described above in Section 6.3.1.

Generic function names are local to the program unit that refers to them. Thus, they can be used for other purposes in other program units.

6.3.3 Intrinsic and Generic Function Usage

Figure 6-2 shows the use of intrinsic and generic function names. In this figure, a single executable program uses the name SIN in four distinct ways:

- As the name of a statement function
- As a generic function name
- As a specific intrinsic function name
- As a user-defined function

Using the name in these four ways emphasizes the local and global properties of the name.

In Figure 6-2, the parenthetical notes are keyed to the notes that follow the figure.

Table 6-3
Generic Function Name Summary

Generic Name	Data Type of Argument	Data Type of Result
ABS	Integer Real COMPLEX*8 COMPLEX*16	Integer Real REAL*4 REAL*8
AINT, ANINT	Real	Real
NINT	Real	Integer
INT	Integer Real Complex	Integer Integer Integer
REAL	Integer Real Complex	REAL*4 REAL*4 REAL*4
DBLE	Integer Real Complex	REAL*8 REAL*8 REAL*8
QEXT	Integer Real Complex	REAL*16 REAL*16 REAL*16
CMPLX	Integer Real Complex	COMPLEX*8 COMPLEX*8 COMPLEX*8
DCMPLX	Integer Real Complex	COMPLEX*16 COMPLEX*16 COMPLEX*16
MOD, MAX, MIN, SIGN, DIM	Integer Real	Integer Real
EXP, LOG, SIN, COS, SQRT	Real Complex	Real Complex
LOG10, TAN, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, TANH	Real	Real

```
C
C    COMPARE WAYS OF COMPUTING SINE.
C
     PROGRAM SINES
     REAL*8 X, PI
     PARAMETER (PI = 3.14159265358979323 8D0)
     COMMON V(3)
C    DEFINE SIN AS A STATEMENT FUNCTION  (Note 1)
     SIN(X) = COS(PI/2-X)
     DO 10 X = -PI, PI, 2*PI/100
               CALL COMPUT(X)
C    REFERENCE THE STATEMENT FUNCTION SIN  (Note 2)
10   WRITE(6,100) X,V, SIN(X)
100  FORMAT (5F10.7)
     END

C
C
     SUBROUTINE COMPUT(Y)
     REAL*8 Y
C    USE INTRINSIC FUNCTION SIN AS ACTUAL ARGUMENT (Note 3)
     INTRINSIC SIN
     COMMON V(3)
C    GENERIC REFERENCE TO DOUBLE PRECISION SINE (Note 4)
     V(1) = SIN(Y)
C    INTRINSIC FUNCTION SINE AS ACTUAL ARGUMENT (Note 5)
     CALL SUB(REAL(Y),SIN)
     END
C
C
     SUBROUTINE SUB(A,S)
C    DECLARE SIN AS NAME OF USER FUNCTION (Note 6)
     EXTERNAL SIN
C    DECLARE SIN AS TYPE REAL*8 (Note 7)
     REAL*8 SIN
     COMMON V(3)
C    EVALUATE INTRINSIC FUNCTION SIN (Note 8)
     V(2) = S(A)
C    EVALUATE USER DEFINED SIN FUNCTION (Note 9)
     V(3) = SIN(A)
     END

C
C
C    DEFINE THE USER SIN FUNCTION (Note 10)
     REAL*8 FUNCTION SIN(X)
     INTEGER FACTOR
     SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)
     1       - X**7/FACTOR(7)
     END

     INTEGER FUNCTION FACTOR(N)
     FACTOR = 1
     DO 10 I=N, 1, -1
10   FACTOR = FACTOR * I
     END
```

Figure 6-2 Multiple Function Name Usage

Notes:

1 A statement function named SIN is defined in terms of the generic function name COS. Since the argument of COS is double-precision, the double-precision cosine function will be evaluated. The statement function SIN is itself single-precision.

2 The statement function SIN is called.

3 The name SIN is declared intrinsic so that the single-precision, intrinsic sine function can be passed as an actual argument at 5.

4 The generic function name SIN is used to refer to the double-precision sine function.

5 The single-precision intrinsic sine function is used as an actual argument.

6 The name SIN is declared a user-defined function name.

7 The type of SIN is declared double-precision.

8 The single- precision sine function passed at 5 is evaluated.

9 The user-defined SIN function is evaluated.

10 The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

6.3.4 Character and Lexical Comparison Library Functions

Character library functions are functions that take character arguments or return character values; lexical comparison library functions are functions that take character arguments and return logical values.

There are four character functions provided with FORTRAN:

- LEN

  The LEN function returns the length of a character expression. The LEN function has the form

  ```
  LEN(c)
  ```

  where:

  c
  is a character expression. The value returned indicates how many bytes there are in the expression.

- INDEX

  The INDEX function searches for a substring (c2) in a specified character string (c1), and, if it finds the substring, returns the substring's starting position. If c2 occurs more than once in c1, the starting position of the first (leftmost) occurrence is returned. If c2 does not occur in c1, the value zero is returned. The INDEX function has the form

  ```
  INDEX (c1, c2)
  ```

  where:

  c1
  is a character expression specifying the string to be searched for the substring specified by c2.

  c2
  is a character expression specifying the substring for which the starting location is to be determined.

- ICHAR

  The ICHAR function converts a character expression to its equivalent ASCII code and returns the ASCII value. ICHAR has the form

  ```
  ICHAR (c)
  ```

  where:

  c
  is the character to be converted to an ASCII code. If c is longer than one byte, only the value of the first byte is returned; the remainder is ignored.

- CHAR

  The CHAR function converts an ASCII integer value to a character value, and returns the character value. CHAR has the form

  ```
  CHAR (i)
  ```

  where:

  i
  is an integer expression.

Examples illustrating the LEN and INDEX functions follow:

LEN Function Example:

```
          SUBROUTINE REVERSE(S)
          CHARACTER T, S*(*)

          J = LEN(S)
          DO 10 I=1, J/2
              T = S(I:I)
              S(I:I) = S(J:J)
              S(J:J) = T
              J = J-1
      10      CONTINUE

          RETURN
          END
```

INDEX Function Example:

```
          SUBROUTINE FIND_SUBSTRINGS(SUB, S)
          CHARACTER*(*) SUB, S
          CHARACTER*132 MARKS

          I = 1
          MARKS = ' '

      10  J = INDEX(S(I:), SUB)
          IF (J .NE. 0) THEN
              I = I + (J-1)
              MARKS(I:I) = '#'
              I = I+1
              IF (I .LE. LEN(S)) GO TO 10
              ENDIF

          WRITE(6,91) S, MARKS
      91  FORMAT( 2(/1X, A))
          END
```

The four lexical comparison functions provided with FORTRAN are:

- LLT, where LLT(X,Y) is equivalent to (X .LT. Y)
- LLE, where LLE(X,Y) is equivalent to (X .LE. Y)
- LGT, where LGT(X,Y) is equivalent to (X .GT. Y)
- LGE, where LGE(X,Y) is equivalent to (X .GE. Y)

The lexical functions have the form

```
func(c,c)
```

where:

func
is one of the symbolic names: LLT, LLE, LGT, or LGE.

c
is a character expression.

The lexical library functions are guaranteed to make comparisons according to the ASCII collating sequence, even on non-ASCII processors. On VAX-11 systems, the lexical library functions are identical to the corresponding character relationals.

An example of the use of the lexical library functions follows:

```
CHARACTER*10 CH2
IF (LGT(CH2, 'SMITH')) STOP
```

The IF statement in this example is equivalent to:

```
IF (CH2 .GT.  'SMITH') STOP
```

CHAPTER 7

INPUT/OUTPUT STATEMENTS

FORTRAN programs use READ and ACCEPT statements for input, and WRITE, REWRITE, TYPE, and PRINT statements for output.

Some forms of these input and output statements translate data from internal (binary) form to external (readable character) form, or vice versa.

READ, WRITE, and REWRITE statements refer explicitly to a unit (a file or device) from or to which data is to be transferred. This unit is called a logical unit and may be connected to a device or file by an OPEN statement (see Section 9.1). ACCEPT, TYPE, and PRINT statements however, do not refer explicitly to a logical unit; instead, they refer implicitly to a default logical unit. The ACCEPT statement is normally connected to the default input device, and the TYPE and PRINT statements are normally connected to the default output device.

The various forms of input/output (I/O) statements can be grouped into the following categories:

- Sequential I/O. Sequential input and output statements transfer records sequentially from and to files, or from and to an I/O device such as a terminal.

- Direct Access I/O. Direct access input and output statements transfer records selected by record number from and to direct-access files.

- Indexed I/O. Indexed input statements transfer from indexed files records selected by data values (keys) contained in the records. Indexed output statements transfer records to indexed files.

- Internal I/O. Internal input and output statements transfer data between variables and arrays defined within a program.

The I/O statement forms can be classified as formatted, list-directed, or unformatted.

Formatted I/O statements contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.

List-directed I/O statements are similar to formatted statements in function, but differ in that they use data types instead of explicit format specifiers to control the translation of data from one form to the other.

Unformatted I/O statements do not contain format specifiers of any kind and therefore are not used to translate the data being transferred. Unformatted I/O is especially appropriate where the data to be output will subsequently be used as input, because unformatted I/O saves execution time by eliminating the data translation process, preserves greater precision in the external data, and usually conserves file storage space.

Table 7-1 shows the various I/O statements, by category, that can be used in FORTRAN programs.

Table 7-1
Available I/O Statements

Statement Name	Statement Category									
	Sequential			Direct		Indexed		Internal		
	F	L	U	F	U	F	U	F		
READ	X	X	X	X	X	X	X	X		
WRITE	X	X	X	X	X	X	X	X		
REWRITE	-	-	-	-	-	X	X	-		
ACCEPT	X	X	-	-	-	-	-	-		
TYPE	X	X	-	-	-	-	-	-		
PRINT	X	X	-	-	-	-	-	-		

KEY

F - Formatted
L - List-Directed
U - Unformatted

I/O statements transfer all data as records -- that is, that which is read or written is a record. The amount of data that one of these records can contain depends on whether unformatted or formatted I/O is used to transfer the data. With unformatted I/O, the I/O statement alone specifies the amount of data to be transferred; with formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Normally, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for formatted I/O statements to transfer data from or to more than one record.

7.1 I/O PROCESSING

The following sections describe in general terms the elements of FORTRAN I/O processing: records, files, internal files, and access modes. See the VAX-11 FORTRAN User's Guide for more detail about FORTRAN I/O processing.

7.1.1 **Records**

A record is a collection of data items, called fields, that are logically related and are processed as a unit. Each FORTRAN I/O statement transfers one record. Formatted I/O statements may transfer additional records.

If an input statement does not use all of the data fields in a record, the remaining fields are ignored. If an input statement requires more data fields than the record contains, either an error condition occurs or, in the case of formatted input, all fields are read as spaces.

If an output statement attempts to write more data fields than the record can contain, an error condition occurs. If an output statement transfers less data than required to fill a fixed-length record, the record is filled with spaces (if a formatted record) or zeroes (if an unformatted record).

7.1.2 **Files**

A file is a collection of logically related records that are arranged in a specific order and treated as a unit. The arrangement or organization of a file is determined when the file is created.

There are three kinds of file organization: sequential, relative, and indexed.

Files are normally stored on disk, though sequential files may also be stored on magnetic tape. Other peripheral devices, such as terminals, card readers, and line printers, are treated as sequential files.

7.1.2.1 **Sequential Organization** - Records in a sequential file are ordered in physical sequence. Each record, except the first, has another record preceding it, and each record, except the last, has another record following it. The physical order in which records appear is usually identical to the order in which the records were originally written to the file.

7.1.2.2 **Relative Organization** - Records in a relative file consist of a specified number of fixed-length cells ordered in physical sequence. These cells are numbered from 1 (the first) to n (the last), with each number representing the location of a record relative to the beginning of the file. Each cell either contains a single record or is empty. The cell number (record number) is used to refer to specific records in the file.

7.1.2.3 **Indexed Organization** - Records in an indexed file are ordered by fields in the records designated as keys. A key is a data field in the record of an indexed file.

When creating an indexed file, you decide which fields in the file's records are to be keys; the contents of these fields are then used to identify specific records for subsequent processing. The length of a key field, and its relative position in the record, are identical for all records in the file.

You must define at least one key for an indexed file. This mandatory key is called the primary key of the file, and usually has a unique value for each record (this is the default condition). You may also define other keys, up to 254 of them, called alternate keys. An alternate key consists of a field that is held in common by, and located in the same position in, each record in the file. Both primary keys and alternate keys may be used to identify a record for retrieval. An alternate key need not have a unique value in each record.

7.1.3 Internal Files

An internal file is not really a file at all, but rather is designated internal storage space that is manipulated to facilitate internal I/O. Its use with formatted sequential READ and WRITE statements reduces the need to use the ENCODE and DECODE statements for internal I/O (see Appendix A).

An internal file consists of a character variable, a character array element, a character array, or a character substring; a record in an internal file consists of any of the above except a character array.

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the variable, array element, or substring. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined -- that is, a value has been assigned to the record.

Prior to data transfer, an internal file is always positioned at the beginning of the first record.

7.1.4 Access Modes

Access mode is the method a program uses to retrieve and store records in a file. The access mode is specified as part of each I/O statement. VAX-11 FORTRAN supports three access modes: sequential, direct, and keyed.

Table 7-2 shows the valid access modes for each file organization.

Table 7-2
Access Modes for Each File Organization

File Organization	Access Mode		
	Sequential	Direct	Keyed
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

1. Records must be fixed-length.

7.1.4.1 **Sequential Access** - Sequential access means that records are processed in sequence. For a sequential file, the sequence is the physical sequence of the records. For a relative file, the sequence is the cell number sequence. For an indexed file, the sequence is in ascending order of key values. If two records have the same key value, the sequence is the order of insertion in the file.

7.1.4.2 **Direct Access** - Direct access means that the program specifies the order of processing by including a direct access record number in each I/O statement. For sequential files, the records must be fixed-length.

7.1.4.3 **Keyed Access** - Keyed access means that the program specifies the order of processing by including a key specification (see Section 7.2.1.5) in an I/O statement to locate the desired record. Your program can mix keyed access and sequential access I/O statements on the same file. Sequential I/O statements access records with increasing key values in the current key-of-reference. You can use keyed I/O statements to position the file to a particular record, and then use sequential I/O statements to process successive records.

7.2 I/O STATEMENT COMPONENTS

I/O statements are comprised of three basic components: the statement keyword, the control list, and the I/O list.

There are six basic statement keywords: READ, ACCEPT, WRITE, REWRITE, TYPE, and PRINT. The first two of these represent input operations, and the remaining four output operations.

The control list and the I/O list are discussed below.

7.2.1 The Control List

The control list of an I/O statement is a list of one or more parameters that control the following functions:

- Designating the logical unit to be acted upon
- Designating the internal file to be acted upon
- Designating whether formatting is to be used for data editing, and if it is, of the format specification
- Designating the number of a direct-access record to be accessed
- Designating the key and key-of-reference of a keyed-access record to be accessed
- Indicating the completion status of an I/O operation
- Transferring control in the event of an error or end-of-file condition

The type of a statement can always be determined by the contents of its control list. For example, the control list of a formatted I/O statement always contains a format specifier (FMT=f or f), whereas that of a list-directed I/O statement always contains an asterisk in place of a format specifier.

The control list has the form

```
(p[,p]...)
```

where:

p
is of the form: keyword = value. The keywords and values are explained in the following sections.

7.2.1.1 **Logical Unit Specifier** - The logical unit specifier is a parameter that specifies the logical unit that is to be accessed. It has one of the following forms:

```
[UNIT=]u
[UNIT=]*
```

where:

u
is an integer expression with a value in the range 0 through 99 that refers to a specific file or I/O device. If necessary, the value is converted to integer data type before being used.

*
specifies that a default input or output unit is to be accessed.

The keyword UNIT= is optional only if the logical unit specifier is the first parameter in the control list.

A logical unit number is connected to a file or device in one of two ways:

- Explicitly through an OPEN statement (see Section 9.1).
- Implicitly by the system. The VAX-11 FORTRAN User's Guide describes the use of implicit connections to logical units in greater detail.

To simplify matters somewhat, you may avoid using the * specification in formatted READ and WRITE statements by substituting an equivalent statement:

```
READ  f, iolist for READ(UNIT=*,FMT=f)iolist

PRINT f,iolist for WRITE(UNIT=*,FMT=f)iolist
```

7.2.1.2 **Internal File Specifier** - An internal file specifier is a parameter that specifies the internal file to be used.

The internal file specifier has the form

```
[UNIT=]cv
```

where:

cv
is the name of a character variable, character array, character array element, or character substring.

The external logical unit specifier and the internal file specifier are mutually exclusive. The keyword UNIT= is optional if the internal file specifier is the first parameter in the control list.

See Section 7.1.3 for more information on internal files.

7.2.1.3 **Format Specifiers** - The format specifier is a parameter that specifies whether explicit or list-directed formatting is to be used and, in the case of explicit formatting, identifies the parameter that will control the formatting. The format specifier has the form

```
[FMT=]f
[FMT=]*
```

where:

f
is the statement label of a FORMAT statement, an integer variable that has been assigned (with an ASSIGN statement) a FORMAT statement label value, the name of an array or array element, or a character expression containing a run-time format.

*
specifies list-directed formatting.

The keyword FMT= is optional only if the format specifier is the second parameter in the control list, and the first parameter is a logical unit or internal file specifier without the optional keyword UNIT=.

Chapter 8 describes FORMAT statements. Section 8.6 describes the interaction between formats and I/O statements.

In sequential I/O statements, you can use an asterisk instead of a format specifier to denote list-directed formatting. See Sections 7.4.1.2 and 7.5.1.2.

7.2.1.4 **Record Specifier** - The record specifier is a parameter that specifies the number of the direct-access record to be accessed. The record specifier has the forms

```
REC= r
'r
```

where:

r
is a numeric expression with a value that represents the position in a direct-access file of the record to be accessed. The value must be greater than or equal to 1, and less than or equal to the maximum number of record cells allowed in the file. If necessary, a record number is converted to integer data type before being used.

7.2.1.5 **Key Specifier** - The key specifier is a parameter that specifies the key of the indexed-file record to be accessed. A key specifier has two components:

- A key expression, which specifies the value of the key to be used.
- A match criterion, which specifies the selection conditions.

A key specifier has one of the following forms:

```
KEY=ke
KEYEQ=ke
KEYGE=ke
KEYGT=ke
```

where:

ke
is a key expression (see below).

Two types of key expressions are supported:

- Character key expressions
- Integer key expressions

Character key expressions must be used with character keys, and integer key expressions must be used with integer keys. A character key expression is an ASCII string in one of the following forms:

- A character expression
- A BYTE (LOGICAL*1) array name containing Hollerith data

The length of the character key expression is the length of the character value or the length of the BYTE array. If the length of the key expression is greater than the length of the key field, an error occurs. If the length of the key expression is less than the length of the key field, then a generic key search is made (see below) rather than an exact key search.

An integer key expression is an integer expression. Real and complex values are not permitted.

The match criterion specifies which key values in the record can match the key expression. There are three possible criteria:

- Equal. The key value must equal the key expression specified.
- Greater. The key value must be greater than the key expression specified.
- Greater than or equal. The key value must be greater than or equal to the key expression specified.

The following parameters are used to establish the desired match criterion:

```
KEY=val  }
KEYEQ=val} specify an equal match.
KEYGT=val specifies a greater match.
KEYGE=val specifies a greater or equal match.
```

For character keys, the comparison is made according to the ASCII collating sequence.

For integer keys, the comparison is made according to the signed integer value.

For character keys, either generic match or exact match can be used. Generic match applies if the key expression in the I/O statement is shorter than the key field in the record. In this case, only the leftmost characters of the key field are used for the match.

For example, if the key expression is 'ABCD' and the key field is ten characters long, then an equal match is obtained for the first record containing 'ABCD' as the first four bytes of the key. The remaining six characters are arbitrary.

Approximate-generic match occurs when approximate match (KEYGT or KEYGE) is selected in addition to generic match. In that case, only the leftmost characters are used for comparison.

For example, if the key expression is 'ABCD', and the key field is five characters long, and a greater-than match is selected, then the value 'ABCDA' does not match; 'ABCEx' does match.

7.2.1.6 **Key-of-Reference Specifier** - The key-of-reference specifier may optionally be included with a key specifier; it is used to specify the index that is to be searched for the specified key value. The key-of-reference specifier has the form

```
KEYID=kn
```

where:

kn
is an integer expression, called the key-of-reference number, that designates the index to be searched.

The key-of-reference number is an integer value in the range zero to the maximum key number defined for the file. A value of zero specifies the primary key, a value of 1 specifies the first alternate key, and so forth.

If no key-of-reference number is given, the key-of-reference is unchanged from the last specification given in a keyed I/O statement for that logical unit.

7.2.1.7 **Input/Output Status Specifier** - The input/output status specifier stores a value in a variable that indicates whether an error or end-of-file condition exists. If the value is zero, no error or end-of-file condition exists. If the value is positive, an error condition exists. If the value is negative, an end-of-file condition exists, but an error condition does not.

The input/output status specifier has the form

```
IOSTAT=ios
```

where:

ios
is an integer variable or array element.

Refer to the VAX-11 FORTRAN User's Guide for more information on the error numbers returned by IOSTAT.

7.2.1.8 **The Transfer-of-Control Specifiers** - The transfer-of-control specifiers are parameters that transfer control of the program to a specific statement in the event of an end-of-file condition or an error condition. The transfer-of-control specifiers have the form

```
END=s

ERR=s
```

where:

s
is the label of the executable statement to which control is to be transferred.

A sequential READ statement can include either or both of the above specifications, in any order. Direct-access READ, keyed-access READ, WRITE, and REWRITE statements can only include the ERR=s specification.

The statement label in the END=s or ERR=s specification must refer to an executable statement within the same program unit as that of the I/O statement.

An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-file record produced by the ENDFILE statement (see Section 9.6) is encountered. End-of-file conditions do not occur in direct-access or keyed-access READ statements. If a READ statement encounters an end-of-file condition during an I/O operation, it transfers control to the statement named in the END=s specification; if there is no END=s specification, an error occurs.

If a READ, WRITE, or REWRITE statement encounters an error condition during an I/O operation, it transfers control to the statement whose label appears in the ERR=s specification. If neither an ERR= specifier nor an IOSTAT= specifier is present, the I/O error terminates program execution.

The VAX-11 FORTRAN User's Guide describes system subroutines that you can use to control error processing. To obtain information from the I/O system on the type of error that occurred, use the IOSTAT parameter discussed in Section 7.2.1.7.

Examples of I/O statements follow:

```
READ (8,END=550) (MATRIX(K),K=1,100)
```

This statement transfers control to statement 550 if an end-of-file condition occurs on logical unit 8.

```
WRITE (6,50,ERR=390)
```

This statement transfers control to statement 390 if an error occurs in the execution of the WRITE statement.

```
READ (1,FORM,ERR=150,END=200) ARRAY
```

This statement transfers control to statement 150 if an error occurs in the execution of the READ statement, and to statement 200 if the end-of-file condition occurs.

7.2.2 Input/Output List

The I/O list in an input or output statement contains the names of variables, arrays, array elements, and character substrings from which or to which data will be transferred. The I/O list in an output statement can also contain constants and expressions to be output.

An I/O list has the following form

```
s[,s]...
```

where:

s
is a simple list element or an implied DO list.

The I/O statement assigns values to, or transfers values from, the list elements in the order in which they appear, from left to right.

7.2.2.1 **Simple List Elements** - A simple I/O list element can be a single variable, array, array element, character substring reference, constant, or expression. For example:

```
WRITE (5,10) J, K(3), 4, (L+4)/2, N
```

When you use an unsubscripted array name in an I/O list, an input statement reads enough data to fill every element of the array; an output statement writes all the values in the array. Data transfer begins with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, the following defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY, with no subscripts, appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

In an input statement, variables in the I/O list can be used in array subscripts later in the list. For example:

```
      READ (1,1250) J,K,ARRAY(J,K)
1250  FORMAT (I1,1X,I1,1X,F6.2)
```

The input record contains the following values:

```
1,3,721.73
```

When the READ statement is executed, the first input value is assigned to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Then the value 721.73 is assigned to ARRAY(1,3). Variables that are to be used as subscripts in this way must appear before (to the left of) their use as the array subscripts in the I/O list.

An output statement I/O list may contain any valid expression. However, this expression must not attempt any further I/O operations on the same logical unit. For example, an output statement I/O list expression must not refer to a function subprogram that performs I/O on the same logical unit.

An input statement I/O list must not contain an expression, except as a subscript expression in an array reference or as an expression in a substring reference.

7.2.2.2 **Implied DO Lists** - An implied DO list is an I/O list element that functions as though it were a part of an I/O statement within a DO loop. Implied DO lists can be used to:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array elements in a sequence different from the order of subscript progression

An implied DO list has the form

```
(list,i=e1,e2[,e3])
```

where:

list
is an I/O list.

i
is an integer or real variable.

e1,e2,e3
are arithmetic expressions.

The variable i and the parameters e1, e2, and e3 have the same forms and the same functions that they have in the DO statement (see Section 4.3). The list immediately preceding the DO loop parameter is the range of the implied DO loop. Elements in that list can reference i, but they must not alter the value of i. Some examples are:

```
WRITE (3,200) (A,B,C, I=1,3)
```

The statement in this example behaves as though you had written:

```
WRITE (3,200) A,B,C,A,B,C,A,B,C
```

Another example is:

```
WRITE (6) (I,(J,P(I),Q(I,J),J=1,L),I=1,M)
```

The I/O list in this example consists of an implied DO list containing another implied DO list nested within it. The implied DO lists together will write a total of (1+3*L) *M fields, varying the Js for each value of I.

In a series of nested implied DO lists, the parentheses indicate the nesting (see Section 4.3.1.2). Execution of the innermost lists is repeated most often. For example:

```
      WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
150   FORMAT (F10.2)
```

Because the inner DO loop is executed 10 times for each iteration of the outer loop, the second subscript, L, advances from 1 through 10 for each increment of the first subscript. This is the reverse of the order of subscript progression. In addition, K is incremented by 2, so only the odd-numbered rows of the array are output.

The entire list of an implied DO list is transmitted before the control variable is incremented. For example:

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

In this example, P(1), Q(1,1), Q(1,2) ...,Q(1,10) are read before I is incremented to 2.

When processing multidimensional arrays, you can use a combination of fixed subscripts and subscripts that vary according to an implied DO list. For example:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

This statement assigns input values to BOX(1,1) through BOX(1,10) and then terminates without affecting any other element of the array.

The value of the control variable can also be output directly. For example:

```
WRITE (6,1111) (I, I=1,20)
```

This statement simply prints the integers 1 through 20.

7.3 SYNTACTICAL RULES

The FORTRAN I/O statements described in Sections 7.4 through 7.8 are subject to the following syntactical rules.

1. When in keyword form, the control-list parameters can appear in any order in the control list, subject to the provisions of Rules 2 and 3.

2. The nonkeyword form of either the logical unit specifier or the internal file specifier must occupy the first (leftmost) position in the control list.

3. When used with a logical unit specifier or internal file specifier, the nonkeyword form of the format specifier must occupy the second position in the control list; the unit or internal file specifier must also be in nonkeyword form (and therefore occupy the first position in the control list).

4. The nonkeyword form of the direct-access record specifier must immediately follow the nonkeyword form of the logical unit specifier.

READ

7.4 THE READ STATEMENTS

The READ statements transfer input data to internal storage from records contained in external logical units, or to internal storage from internal files. There are four categories of READ statements: sequential, direct-access, indexed, and internal.

7.4.1 The Sequential READ Statements

Sequential READ statements transfer input data to internal storage from external records accessed under the sequential mode of access. There are three classes of sequential READ statements: formatted, list-directed, and unformatted.

The forms of the three classes of sequential READ statement are as follows:

Formatted Sequential READ Statements

```
READ(extu, fmt [,iostat][,err][,end])[list]

READ f[,list]
```

List-Directed READ Statements

```
READ (extu, * [,iostat] [,err] [,end]) [list]

READ *[, list]
```

Unformatted Sequential READ Statements

```
READ (extu [,iostat] [,err] [,end]) [list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

fmt
is a format specifier. See Section 7.2.1.3.

f
is the nonkeyword form of a format specifier. See fmt, above.

*
specifies list-directed formatting.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
end
are transfer-of-control specifiers. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.1.1 **The Formatted Sequential READ Statement** - The formatted sequential READ statement:

- Reads character data from one or more external records accessed under the sequential or keyed mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

If the number of I/O list elements in a statement is less then the number of fields in an input record, the statement ignores the excess fields.

7.4.1.2 **The List-Directed READ Statement** - The list-directed READ statement:

- Reads character data from records accessed under the sequential mode of access

- Translates the data from external to binary form using the data types of the elements in the I/O list, and the forms of the data, to provide editing
- Assigns the translated data to the elements in the I/O list in the order, from left to right, in which those elements appear in the list

The external records from which list-directed READ statements read data contain a sequence of values and value separators.

A value in one of these records may be any one of the following:

- A constant
- A null value
- A repetition of constants in the form r*c
- A repetition of null values in the form r*

Each constant has the form of the corresponding FORTRAN constant. A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Spaces can occur between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis. A logical constant represents true or false values -- that is, .TRUE. or any value beginning with T, .T, t, or .t; or .FALSE. or any value beginning with F, .F, f, or .f. A character constant is delimited by apostrophes, with an apostrophe occurring within a character constant being represented by two consecutive apostrophes. Hollerith, octal, and hexadecimal constants are not permitted.

A null value is specified by two consecutive commas with no intervening constant, or by an initial comma or trailing comma. Spaces can occur before or after the commas. A null value indicates that the corresponding list element remains unchanged. Also, a null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

The form r*c indicates r occurrences of c, where r is a nonzero, unsigned integer constant and c is a constant. Spaces are not permitted except within the constant c as specified above.

The form r* indicates r occurrences of a null value, where r is an unsigned integer constant.

A value separator in a record may be any one of the following:

- One or more spaces or tabs
- A comma, with or without surrounding spaces or tabs
- A slash, with or without surrounding spaces or tabs

The slash terminates processing of the input statement and the record, leaving all remaining I/O list elements unchanged.

When any of the above appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a space character except when it occurs in a character constant. In this case, the end of the record is ignored and the character constant is continued with the next record. That is, the last character in the previous record is followed immediately by the first character of the next record.

Spaces at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the spaces at the beginning of the record are considered part of the constant.

Input constants can be any of the following data types: integer, real, logical, complex, and character. The data type of the constant determines the data type of the value and the translation from external to internal form.

A numeric list element can correspond only to a numeric constant, and a character list element can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see Table 3-1).

Each input statement will read one or more records as required to satisfy the I/O list. If a slash separator occurs or the I/O list is exhausted before all the values in a record are used, the remainder of the record is ignored.

An example of the use of list-directed READ statements follows:

A program unit consists of:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
     .
     .
     .
```

And the external record to be read contains:

```
4 6.3 (3.4,4.2), (3, 2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

Upon execution of the program unit, the following values are assigned to the I/O list elements:

I/O List Element	Value
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI'JK

A, B, and J are unchanged.

7.4.1.3 **The Unformatted Sequential READ Statement** - The unformatted sequential READ statement reads an external record accessed under the sequential or keyed mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted sequential READ statement reads exactly one record. If the I/O list does not use all the values in a record -- that is, there are more values in the record than elements in the list -- the remainder of the record is discarded. If the number of list elements is greater than the number of values in the record, an error occurs.

If a statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

The unformatted sequential READ statement can only read records created by unformatted sequential WRITE statements.

Some examples of the use of the unformatted sequential READ statement follow:

```
READ(UNIT=1) FIELD1, FIELD2
```

In this example, the READ statement reads one record from logical unit 1 and assigns values of binary data to variables FIELD1 and FIELD2, in that order.

```
READ (8)
```

In this example, the READ statement advances Logical Unit 8 one record.

7.4.2 The Direct-Access READ Statements

Direct-Access READ statements transfer input data to internal storage from external records accessed under the direct mode of access. There are two classes: formatted and unformatted.

The forms of the two classes of direct-access READ statement are as follows:

Formatted Direct-Access READ Statements

```
READ( extu, rec, fmt [,iostat][,err])[list]
```

Unformatted Direct-Access READ Statements

```
READ( extu, rec [,iostat][,err])[list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

rec
is a record specifier. See Section 7.2.1.4.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
is a transfer-of-control specifier. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.2.1 **The Formatted Direct-Access READ Statement** - The formatted direct-access READ statement:

- Reads character data from one or more external records accessed under the direct mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

If the I/O list and formatting do not use all the characters in a record, the remainder of the record is discarded; if the I/O list and formatting require more characters than are contained in the record, the remaining fields are read as spaces.

An example of the use of the formatted direct-access READ statement follows:

```
      READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
10    FORMAT (10I2)
```

In this example, the READ and FORMAT statements read the first ten fields from record 35 in logical unit 2, translate the values to binary form, and then assign the translated values to the internal storage locations of the ten elements of the array NUM.

7.4.2.2 **The Unformatted Direct-Access READ Statement** - The unformatted direct-access READ statement reads an external record accessed under the direct mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by that element's data type.

The unformatted direct-access READ statement reads exactly one record. If that record contains more fields than there are elements in the I/O list of the statement, the unused fields are discarded; if there are more elements than fields, an error occurs.

Examples of the use of unformatted direct-access READ statements follow:

```
READ (1'10) LIST(1), LIST(8)
```

In this example, the READ statement reads record 10 in logical unit 1 and assigns binary integer values to elements 1 and 8 of the array LIST.

```
READ (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

In this example, the READ statement reads record 58 in logical unit 4 and assigns binary values to 5 elements of the array RHO.

7.4.3 Indexed READ Statements

The indexed READ statement transfers input data to internal storage from external records accessed under the keyed mode of access. There are two classes: formatted and unformatted.

A series of records in an indexed file may be read in key-of-reference sequence by using a sequential READ statement in conjunction with an indexed READ statement. The first record in the sequence is found using the indexed statement, the rest using sequential statements.

The forms of the two classes of indexed READ statement are as follows:

Formatted Indexed READ Statement

```
READ( extu, fmt, key [,keyid] [,iostat][,err])[list]
```

Unformatted Indexed READ Statement

```
READ( extu, key [,keyid] [,iostat][,err])[list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

fmt
is a format specifier. See Section 7.2.1.3.

key
is a key specifier. See Section 7.2.1.5.

keyid
is a key-of-reference specifier. See Section 7.2.1.6.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
end
are transfer-of-control specifiers. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.4.3.1 **The Formatted Indexed READ Statement** - The formatted indexed READ statement:

- Reads character data from one or more external records accessed under the keyed mode of access
- Translates the data from character to binary form using format specifications to provide editing
- Assigns the translated values to the elements in the I/O list, in the order, from left to right, in which they appear in the list

The formatted indexed READ statement may only be used on indexed files. If the I/O list and format specifications specify that additional records are to be read, the statement reads those additional records sequentially using the current key-of-reference value.

If the KEYID parameter is omitted, the key-of-reference remains unchanged from the most recent specification.

If the specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the key value is longer than the key field referred to, an error occurs.

An example of the use of the formatted indexed READ statement follows:

```
READ(3,KAT(25),KEY='ABCD') A,B,C,D
```

In this example the READ statement retrieves a record with the value of 'ABCD' in the primary key, and then uses the format contained in the array item KAT(25) to read the first four fields from the record into variables A,B,C, and D.

7.4.3.2 **The Unformatted Indexed READ Statement** - The unformatted indexed READ statement reads an external record accessed under the keyed mode of access; it assigns the fields of binary data contained in that record to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list. The data is not translated. The amount of data assigned to each element is determined by the element's data type.

The unformatted indexed READ statement reads exactly one record, and may only be used on indexed files. If the number of I/O list elements is less than the number of fields in the record being read, the unused fields in the record are discarded. If the number of I/O list elements is greater than the number of fields, an error occurs.

If a specified key value is shorter than the key field referred to, the key value is matched against the leftmost characters of the appropriate key field until a match is found; the record supplying the match is then read. If the specified key value is longer than the key field referred to, an error occurs.

Some examples of the use of the unformatted indexed READ statement follow:

```
OPEN (UNIT=3, STATUS='OLD',
1         ACCESS='KEYED', ORGANIZATION='INDEXED',
2         FORM='UNFORMATTED',
3         KEY=(1:5, 30:37, 18:23))

READ (3,KEY='SMITH') ALPHA,BETA
```

In this example, the READ statement reads from the file connected to logical unit 3 and retrieves the record with the value 'SMITH' in the primary key field (bytes 1 to 5). The first two fields of the record retrieved are placed in variables ALPHA and BETA, respectively.

```
READ (3,KEYGE='XYZDEF',KEYID=2,ERR=99) IKEY
```

In this example, the READ statement retrieves the first record having a value equal to or greater than 'XYZDEF' in the second alternate key field (bytes 18 to 23). The first field of that record is placed in the variable IKEY.

7.4.4 The Internal READ Statement

The internal READ statement transfers input data to internal storage from an internal file.

The DECODE statement discussed in Appendix A may be used as an alternative to the internal READ statement.

The internal READ statement is always formatted, and has the following form

```
READ (intu, fmt[,iostat][,err][,end])[list]
```

where:

intu
is an internal file specifier. See Section 7.2.1.2.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
end
are transfer-of-control specifiers. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

The internal READ statement:

- Reads character data from an internal file
- Translates the data from character to binary form using format specifications to provide editing

- Assigns the translated data to the elements in the I/O list, in the order, from left to right, in which those elements appear in the list

Refer to Section 7.1.3 for information on the characteristics and use of internal files.

The following program segment demonstrates the use of internal-file reads:

```
      INTEGER IVAL
      CHARACTER TYPE, RECORD*80
      CHARACTER * (*) AFMT, IFMT, OFMT, ZFMT
      PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',
     1     ZFMT= '(Z8)')
      ACCEPT AFMT, ILEN, RECORD
      TYPE= RECORD (1:1)
      IF (TYPE .EQ. 'D') THEN
          READ (RECORD (2:MIN(ILEN, 11)), IFMT)IVAL
      ELSEIF (TYPE .EQ. '0') THEN
          READ (RECORD (2:MIN(ILEN, 12)), OFMT)IVAL
      ELSEIF (TYPE .EQ. 'X') THEN
          READ (RECORD (2:MIN(ILEN, 9)),ZFMT)IVAL
      ELSE
          PRINT *, 'ERROR'
      ENDIF
      END
```

This program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal-file reads to make appropriate conversions from character string representations to binary.

WRITE

7.5 THE WRITE STATEMENTS

The WRITE statements transfer output data from internal storage to records contained in user-specified external logical units, or from internal storage to internal files. There are four categories of WRITE statements: sequential, direct-access, indexed, and internal.

WRITE statements cannot write to existing records in an indexed file. For statements that can perform this function in indexed files, refer to the REWRITE statement discussed in Section 7.6.

7.5.1 The Sequential WRITE Statements

Sequential WRITE statements transfer output data from internal storage to external records accessed under the sequential mode of access. There are three classes of sequential WRITE statements: formatted, list-directed, and unformatted.

The forms of the three classes of sequential WRITE statement are as follows:

Formatted Sequential WRITE Statements

```
WRITE( extu,fmt [,iostat] [,err]) [list]
```

List-Directed WRITE Statements

```
WRITE( extu, * [,iostat][,err]) [list]
```

Unformatted Sequential WRITE Statement

```
WRITE( extu [,iostat] [,err]) [list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
is a transfer-of-control specifier. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.1.1 **The Formatted Sequential WRITE Statement** - The formatted sequential WRITE statement:

- Reads specified data from internal storage
- Translates the data from binary to character form using format specifications to provide editing
- Writes the translated values to an external record accessed under the sequential mode of access

The length of the records written to a user-specified output device (for example, a line printer) must not exceed the maximum record length which that device can process. In the case of a line printer, this maximum is usually 132 characters.

Using an appropriate format specification, a statement can write more than one record.

Because numeric data transferred by formatted output statements is always rounded during its conversion from binary to character form, a loss of precision may result if this data is subsequently used as input. It is recommended, therefore, that whenever numeric output is to be used subsequently as input, unformatted output and input statements be used for data transfer.

Some examples of the use of formatted sequential WRITE statements follow:

```
      WRITE (6,650)
650   FORMAT (' HELLO THERE')
```

In this example, the WRITE statement writes one record, consisting of the contents of the character constant in the format statement, to logical unit 6.

```
      WRITE (1,95) AYE,BEE,CEE
95    FORMAT (3F8.5)
```

In this example, the WRITE statement writes one record consisting of fields AYE, BEE, and CEE to logical unit 1.

```
      WRITE (1,900) DEE,EEE,EFF
900   FORMAT (F8.5)
```

In this example, the Write statement writes three separate records to logical unit 1; each record consists of only one field.

7.5.1.2 **The List-Directed WRITE Statement** - The list-directed WRITE statement:

- retrieves specified data from internal storage
- translates that data from binary to character form using the data type of the elements in the I/O list to provide editing, and
- writes the translated values to an external record accessed under the sequential mode of access.

The values transferred as output by the list-directed WRITE statement have the same forms as the constant values transferred as input by the list-directed READ and ACCEPT statements, with the following exception: Character constants are transferred without delimiting apostrophes, and each internal apostrophe is represented by only one apostrophe instead of two. As a consequence of this exception, records containing list-directed character output data can be printed but cannot be used for list-directed input. (Refer to Section 7.4.1.2 for a full discussion on list-directed value forms.)

Table 7-3 below shows the default output formats for each data type.

Table 7-3
List-Directed Output Formats

Data Type	Output Format
LOGICAL*1	I5
LOGICAL*2	L2
LOGICAL*4	L2
INTEGER*2	I7
INTEGER*4	I12
REAL	1PG15.7
REAL*8	1PG25.16
REAL*16	1PG42.33E3
COMPLEX	1X,'(',1PG14.7, ',', 1PG14.7,')'
COMPLEX*16	1X,'(',1PG25.16,',',1PG25.16,')'
CHARACTER	1X, An (where n is the length of the character expression.)

Note that:

- List-directed output statements do not produce octal values, null values, slash separators, or repeated forms of values.
- List-directed output edits a complex value so that there are no embedded spaces in the value.
- Each output record begins with a space for carriage control.
- Each output statement writes one or more complete records.
- Each individual output value is contained within a single record, with the exception of character constants longer than one record length and complex constants that can be split after the comma.

An example of the use of the list-directed WRITE statement follows:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE(1,*) 'ARRAY VALUES FOLLOW'
WRITE(1,*) A,4
```

In this example, the WRITE statements write the following records to logical unit 1:

```
ARRAY VALUES FOLLOW
   3.400000        3.400000        3.400000        3.400000        4
```

7.5.1.3 **The Unformatted Sequential WRITE Statement** - The sequential unformatted WRITE statement transfers specified binary data from internal storage to an external record accessed under the sequential mode of access. The data are not translated.

The sequential unformatted WRITE statement writes exactly one record; if there is no I/O list, the statement writes one null record.

Some examples of the use of the unformatted sequential READ statement follow:

```
WRITE(1) (LIST(K),K=1,5)
```

In this example, the WRITE statement writes a record to logical unit 1 containing the values, in binary form, of elements 1 through 5 of the array LIST.

```
WRITE(4)
```

In this example, the WRITE statement writes one null record to logical unit 4.

7.5.2 The Direct-Access WRITE Statements

Direct-Access WRITE statements transfer output data from internal storage to external records accessed under the direct mode of access. There are two classes of direct-access WRITE statements: formatted and unformatted.

The OPEN statement is used to establish the attributes of a direct-access file.

The forms of the two classes of direct-access WRITE statements are as follows:

Formatted Direct-Access WRITE Statements

```
WRITE( extu, rec, fmt [,iostat][,err])[list]
```

Unformatted Direct-Access WRITE Statements

```
WRITE( extu, rec [,iostat][,err])[list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

rec
is a record specifier. See Section 7.2.1.4.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output specifier. See Section 7.2.1.7.

err
is a transfer-of-control specifier. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.2.1 **The Formatted Direct-Access WRITE Statement** - The formatted direct-access WRITE statement:

- retrieves binary values from internal storage
- translates those values to character form using format specifications to provide editing and
- writes the translated data to a user-specified external record accessed under the direct mode of access.

If the values specified by the I/O list and formatting do not fill the output record being written, the unused portion of the record is filled with space characters. If the values overfill the record, an error occurs.

7.5.2.2 **The Unformatted Direct-Access WRITE Statement** - The unformatted direct-access WRITE statement retrieves binary values from internal storage and writes those values to a user-specified external record accessed under the direct mode of access. The values are not translated.

If the values specified by the I/O list do not fill the output record being written, the unused portion of the record is filled with zeros. If the values do not fit in the record, an error occurs.

7.5.3 The Indexed WRITE Statements

The indexed WRITE statements transfer output data from internal storage to external records accessed under the keyed mode of access. There are two classes of indexed WRITE statements: formatted and unformatted.

The indexed WRITE statement always writes a new record. The REWRITE statement discussed in Section 7.6 is used to update an existing record.

The OPEN statement is used to establish the attributes of an indexed file.

The syntactic form of the indexed WRITE statement is identical to that of the sequential WRITE statement; the two statements differ only in that the indexed WRITE statement refers to a logical unit connected to an indexed file, whereas the sequential WRITE statement refers to a logical unit connected to a sequential file. The forms of the two classes of indexed WRITE statement are as follows:

Formatted Indexed WRITE Statements

```
WRITE( extu, fmt [,iostat][,err])[list]
```

Unformatted Indexed WRITE Statements

```
WRITE( extu [,iostat][,err])[list]
```

where:

extu
is a logical unit specifier. See Section 7.2.1.1.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output specifier. See Section 7.2.1.7.

err
is a transfer-of-control specifier. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

7.5.3.1 **The Formatted Indexed WRITE Statement** - The formatted indexed WRITE statement:

- retrieves binary values from internal storage
- translates those values to character form using format specifications to provide editing, and
- writes the translated data to one or more external records accessed under the keyed mode of access.

No key parameters are required in the list of control parameters because all necessary key information is contained in the output record.

If the values specified by the I/O list and formatting do not fill a fixed-length record being written, the unused portion of the record is filled with space characters. If additional records are specified, these are inserted in the file logically according to the key values contained in each record.

An example of the use of formatted indexed WRITE statement follows:

```
      WRITE (4,100) KEYVAL (RDATA (I), I=1,20)
100   FORMAT (A10,20F15.7)
```

In this example, the WRITE statement writes the translated values of each of the 20 elements of the array RDATA to a new formatted record in the indexed file connected to logical unit 4.

7.5.3.1 **The Unformatted Indexed WRITE Statement** - The unformatted indexed WRITE statement retrieves binary values from internal storage and writes those values to an external record accessed under the keyed mode of access. The values are not translated.

No key parameters are required in the list of control parameters because all necessary key information is contained in the output record.

If the values specified by the I/O list do not fill a fixed-length record being written, the unused portion of the record is filled with zeros; if the values specified overfill the record, an error occurs.

7.5.4 The Internal WRITE Statement

The internal WRITE statement transfers output data from internal storage to an internal file.

You can also use the ENCODE statement discussed in Appendix C to control internal output.

The internal WRITE statement is always formatted, and has the form

```
WRITE (intu, fmt[,iostat][,err])[list]
```

where:

intu
is an internal file specifier. See Section 7.2.1.2.

fmt
is a format specifier. See Section 7.2.1.3.

iostat
is an input/output status specifier. See Section 7.2.1.7.

err
is a transfer-of-control specifier. See Section 7.2.1.8.

list
is an I/O list. See Section 7.2.2.

Refer to Section 7.3 for the syntactical rules that govern the use of the above parameters.

The internal WRITE statement:

- Retrieves data from internal storage
- Translates that data from binary to character form using format specifications to provide editing
- Writes the translated values to an internal file

Refer to Section 7.1.3 for information on the characteristics and use of internal files.

REWRITE

7.6 THE REWRITE STATEMENT

The REWRITE statement transfers output data from internal storage to the current record in an indexed file. There is only one category of REWRITE statement: indexed.

7.6.1 The Indexed REWRITE Statement

The indexed REWRITE statement transfers output data from internal storage to the last record in an indexed file to be accessed by a READ statement. There are two classes of indexed REWRITE statements: formatted and unformatted.

The OPEN statement is used to establish the attributes of an indexed file.

The forms of the two classes of indexed REWRITE statement are as follows:

Formatted Indexed REWRITE Statement

```
REWRITE( extu,fmt [,iostat][,err]) [list]
```

Unformatted Indexed REWRITE Statement

```
REWRITE( extu [,iostat][,err])[list]
```

where extu, fmt, iostat, err, and list are defined as they are for the indexed WRITE statements discussed in Section 7.5.3. Refer to Section 7.3 for applicable syntactical rules.

7.6.1.1 **The Formatted Indexed REWRITE Statement** - The formatted indexed REWRITE statement:

- retrieves binary values from internal storage
- translates those values to character form using format specifiers to provide editing, and
- writes the translated data to an existing record in an indexed file.

The record written to is the current record in the file--that is, the last record to be accessed by a preceding indexed or sequential READ statement. Note that changing the primary key value usually results in an error, and that attempting to rewrite more than one record causes an error. Any unused space in a rewritten fixed-length record is filled with spaces; if the record is too long, an error occurs.

An example of the use of a formatted indexed REWRITE statement follows:

```
      REWRITE(3, 10, ERR=99) NAME, AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

In this example, the REWRITE statement updates the current record contained in the indexed file connected to logical unit 3 with the values represented by NAME, AGE, and BIRTH.

7.6.1.2 **The Unformatted Indexed REWRITE Statement** - The unformatted indexed REWRITE statement retrieves binary values from internal storage and writes those values to an existing record in an indexed file. The values are not translated.

The record written to is the current record in the file--that is, the last record to be accessed by a preceding indexed or sequential READ statement. Note that changing the primary key value usually results in an error. Any unused space in a rewritten fixed-length record is filled with zeros; if the record is too long, an error occurs.

ACCEPT

7.7 THE ACCEPT STATEMENT

The ACCEPT statement transfers input data to internal storage from external records accessed under the sequential mode of access.

ACCEPT statements can only be used on implicitly connected logical units.

The ACCEPT statement has the following forms:

```
ACCEPT f[,list]

ACCEPT *[,list]
```

where:

f
is the nonkeyword form of a format specifier. See Section 7.2.1.3.

*****
specifies list-directed formatting.

list
is an I/O list. See Section 7.2.2.

The ACCEPT statement functions exactly as the formatted sequential READ statement discussed in Section 7.4.1.1, with the following important exception: The ACCEPT statement can never be connected to user-specified logical units.

An example of the use of the formatted ACCEPT statement follows:

```
      CHARACTER *10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

In this example, the ACCEPT statement reads character data from the implicit unit and assigns binary values to each of the five elements of the array CHARAR.

TYPE
PRINT

7.8 THE TYPE AND PRINT STATEMENTS

The TYPE and PRINT statements transfer output data from internal storage to external records accessed under the sequential mode of access.

TYPE and PRINT statements have the following forms:

```
TYPE f[,list]
PRINT f [,list]

TYPE * [, list]
PRINT * [, list]
```

where:

f
is the nonkeyword form of a format specifier. See Section 7.2.1.3.

*****
specifies list-directed formatting.

list
is an I/O list. See Section 7.2.2.

TYPE and PRINT statements function exactly as the formatted sequential WRITE statement discussed in Section 7.5.1.1, with the following important exception: The formatted sequential TYPE and PRINT statements can never be used to transfer data to user-specified logical units.

An example of the use of a formatted sequential PRINT statement follows:

```
      CHARACTER*16 NAME, JOB
      PRINT 400, NAME, JOB
400   FORMAT ('NAME=', A, 'JOB=', A)
```

In this example, the PRINT statement writes one record to the implicit output device; the record consists of four fields of character data.

CHAPTER 8

FORMAT STATEMENTS

FORMAT statements are nonexecutable statements used with formatted I/O statements and with ENCODE and DECODE statements. A FORMAT statement describes the format in which data is to be transferred, and what data conversion and editing are required to achieve that format.

FORMAT statements have the form

FORMAT ($q_1 f_1 s_1 f_2 s_2 \ldots f_n q_n$)

where:

g
is zero or more slash (/) record terminators.

f
is a field descriptor or a group of field descriptors enclosed in parentheses.

s
is a field separator.

The entire list of field descriptors and field separators, including the parentheses, is called the format specification. The list must be enclosed in parentheses.

A field descriptor in a format specification has one of the following forms:

[r]c [r]cw [r]cw.m [r]cw.d[Ee]

where:

r
is the repeat count for the field descriptor. If you omit r, the repeat count is assumed to be 1.

c
is a format code (I,O,Z,F,E,D,G,L,A,H,X,T,P,Q,$, :,BN,BZ,S,SP,SS,TL, or TR).

w
is the external field width, in characters.

m
is the minimum number of characters that must appear within the field (including leading zeros).

d
is the number of characters to the right of the decimal point.

E
in this context, identifies an exponent field.

e
is the number of characters in the exponent.

The terms r, w, m, and d must all be unsigned integer constants or variable format expressions; r and w must be less than or equal to 32767, and m, d, and e must be less than or equal to 255. The r term is optional; however, you cannot use it in some field descriptors (see Section 8.1.25). The d and e terms are required in some field descriptors and are invalid in others. You are not allowed to use PARAMETER constants for the terms r, w, m, d, or e.

The field descriptors are:

- Integer -- Iw, Ow, Zw, Iw.m, Ow.m, Zw.m
- Logical -- Lw
- Real and complex -- Fw.d, Ew.d, Dw.d, Gw.d, Ew.dEe, Gw.dEe
- Character -- Aw
- Editing, and character and Hollerith constants -- nH, '...', nX, Tn, TLn, TRn, nP, Q, $, :, BN, BZ, S, SP, SS (n is a number of characters or character positions)

Section 8.1 describes each field descriptor in detail.

The first character in an output record generally contains carriage control information (see Section 8.2 for more information).

The field separators are comma and slash. A slash is also a record terminator. Sections 8.3 and 8.4 describe in detail the functions of the field separators.

You can create a format during program execution by using a run-time format instead of a FORMAT statement. Section 8.5 describes run-time formats.

During data transfers, the format specification is scanned from left to right. Data conversion is performed by correlating the elements in the I/O list with the corresponding field descriptors. In H field descriptors and character constants, data transfer takes place entirely between the field descriptor and the external record. Section 8.6 describes in detail the interaction between the format specifier and the I/O list.

Section 8.7 summarizes the rules for writing FORMAT statements.

8.1 FIELD AND EDIT DESCRIPTORS

A field descriptor describes the size and format of a data item or of several data items; each data item in the external medium is called an external field. An edit descriptor specifies an editing function to be performed on a data item or items.

The following sections describe each of the field and edit descriptors in detail.

The numeric field descriptors ignore leading spaces in the external field. But they treat embedded and trailing spaces as zeros unless the BN edit descriptor is in effect, or BLANK = 'NULL' is in effect for the logical unit, in which case all spaces are ignored.

At the beginning of the execution of each formatted input statement, the BLANK attribute for the relevant unit determines the interpretation of spaces; the VAX-11 FORTRAN defaults are BLANK= 'NULL' when an OPEN has been done, and BLANK= 'ZERO' when no explicit OPEN has been done. During the execution of a formatted input statement, the interpretation of spaces may be controlled by BN and BZ edit descriptors -- that is, the default interpretation may be superseded by either of these. The BN and BZ edit descriptors affect only the formatted I/O statement of which they are a part (as do the S, SP, and SS edit descriptors).

8.1.1 **BN Edit Descriptor**

The BN descriptor causes the processor to ignore all the embedded and trailing blanks it encounters within a numeric input field. It has the form

```
BN
```

The effect is that of actually removing the blanks and right-justifying the remainder of the field. A field of all blanks is treated as zero. The BN descriptor affects only I, O, Z, F, E, D, and G editing, and only upon the execution of an input statement.

8.1.2 **BZ Edit Descriptor**

The BZ descriptor causes the processor to treat all the embedded and trailing blanks it encounters within a numeric input field as zeros. It has the form

```
BZ
```

The BZ descriptor affects only I, O, Z, F, E, D, and G editing, and only upon the execution of an input statement.

8.1.3 **SP Edit Descriptor**

An SP descriptor causes the processor to produce a plus character in any position where this character would otherwise be optional. It has the form

```
SP
```

The SS descriptor affects only I,F,E,D, and G editing, and only upon the execution of an output statement.

8.1.4 SS Edit Descriptor

The SS descriptor causes the processor to suppress a leading plus character from any position where this character would normally be produced as an optional character; it has the opposite effect of the SP field descriptor described above. The SS descriptor has the form

SS

The SS descriptor affects only I, F, E, D, and G editing, and only upon execution of an output statement.

8.1.5 S Edit Descriptor

The S edit descriptor reinvokes optional plus characters (+) in numeric output fields. It has the form

S

The S descriptor counters the action of either the SP or SS descriptor by restoring to the processor the discretion of producing plus characters on an optional basis.

The same restrictions apply as do those for the SP and SS descriptors.

8.1.6 I Field Descriptor

The I field descriptor transfers decimal integer values. It has the form

Iw[.m]

The corresponding I/O list element must be of either integer or logical data type.

In an input statement, the I field descriptor transfers w characters from the external field and assigns them to the corresponding I/O list element as an integer value. The external data must have the form of an integer constant; it cannot contain a decimal point or exponent fields.

If the value of the external field exceeds the range of the corresponding list element, an error occurs. If the first nonblank character of the external field is a minus sign, the field is treated as a negative value. If the rirst nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all-blank field is treated as a value of zero.

Input Example:

Format	External Field	Internal Value
I4	2788	2788
I3	-26	-26
I9	△△△△△△312	312

In an output statement, the I field descriptor transfers the value of the corresponding I/O list element, right justified, to an external field w characters long. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width,

the entire field is filled with asterisks. If the value of the list element is negative, the field will have a minus sign as its leftmost, nonblank character. The term w must therefore be large enough to provide for a minus sign, when necessary. If m is present, the external field consists of at least m digits, and is zero-filled on the left, if necessary.

Output Example:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	△△174
I2	3244	**
I3	-473	***
I7	29.812	Not permitted: error
I4.2	1	△△01
I4.4	1	0001

Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

8.1.7 O Field Descriptor

The O field descriptor transfers octal (base 8) values, and can be used with any data type. It has the form

Ow[.m]

In an input statement, the O field descriptor transfers w characters from the external field and assigns them as an octal value to the corresponding I/O list element. The external field can contain only the numerals 0 through 7; it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

Input Example:

Format	External Field	Internal Octal Value
O5	32767	32767
O4	16234	1623
O3	97△	Not permitted: error

In an output statement, the O field descriptor transfers the octal value of the corresponding I/O list element, right justified, to an external field w characters long. No signs are output; a negative value is transmitted in internal form. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.

Output Example:

Format	Internal (Decimal) Value	External Representation
O6	32767	△77777
O6	-32767	100001
O2	14261	**
O4	27	△△33
O5	10.5	41050
O4.2	7	△△07
O4.4	7	0007

Note that if m is zero, and the external representation is zero, the external field is blank-filled.

8.1.8 Z Field Descriptor

The Z field descriptor transfers hexadecimal (base 16) values, and can be used with any data type. It has the form

Zw[.m]

In an input statement, the Z field descriptor transfers w characters from the external field and assigns them as a hexadecimal value to the corresponding I/O list element. The external field can contain only the numerals 0 through 9 and the letters A (a) through F (f); it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

Input Example:

Format	External Field	Internal Hexadecimal Value
Z3	A94	A94
Z5	A23DEF	A23DE
Z5	95.AF2	Not permitted: error

In an output statement, the Z field descriptor transfers the hexadecimal value of the corresponding I/O list element, right justified, to an external field w characters long. No signs are output; a negative value is transmitted in internal form. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.

Output Example:

Format	Internal (Decimal) Value	External Representation
Z4	32767	7FFF
Z5	-32767	△8001
Z2	16	10
Z4	-10.5	C228
Z3.3	2708	A94
Z6.4	2708	△△0A94

Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

8.1.9 F Field Descriptor

The F field descriptor transfers real values. It has the form

Fw.d

The corresponding I/O list element must be of real data type; or it must be either the real or the imaginary part of a complex data type.

In an input statement, the F field descriptor transfers w characters from the external field and assigns them, as a real value, to the corresponding I/O list element. If the first nonblank character of the external field is a minus sign, the field is treated as a negative value. If the first nonblank character is a plus sign, or if no sign appears in the field, the field is treated as a positive value. An all-blank field is treated as a value of zero.

If the field contains neither a decimal point nor an exponent, it is treated as a real number of w digits, in which the rightmost d digits are to the right of the decimal point, with leading zeros assumed, if necessary. If the field contains an explicit decimal point, the location of that decimal point overrides the location specified by the field descriptor. If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

Input Example:

Format	External Field	Internal Value
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

In an output statement, the F field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal positions and right justified, to an external field w characters long. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks.

The term w must be large enough to include: a minus sign when necessary (plus signs are optional); at least one digit to the left of the decimal point; the decimal point; and d digits to the right of the decimal. Therefore, w must be greater than or equal to d+3.

Output Example:

Format	Internal Value	External Representation
F8.5	2.3547188	△2.35472
F9.3	8789.7361	△8789.736
F2.1	51.44	**
F10.4	-23.24352	△△-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

8.1.10 E Field Descriptor

The E field descriptor transfers real values in exponential form. It has the form

Ew.d[Ee]

The corresponding I/O list element must be of real data type; or it must be either the real or the imaginary part of a complex data type.

In an input statement, the E field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F field descriptor interprets and assigns data in exactly the same way.

Input Example:

Format	External Field	Internal Value
E9.3	734.432E3	734432.0
E12.4	△△1022.43E-6	1022.43E-6
E15.3	52.3759663△△△△△	52.3759663
E12.5	210.5271D+10	210.5271E10

Note that, in the last example, the E field descriptor treats the D exponent field indicator as an E indicator if the I/O list element is single-precision.

In an output statement, the E field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal digits and right justified, to an external field w characters long. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks.

When you use the E field descriptor, data output is transferred in a standard form. This form consists of: a minus sign when necessary (plus signs are optional); a zero; a decimal point; d digits to the right of the decimal points; and an e+2-character exponent. The exponent has one of the following forms:

E+nn
E-nn } Ew.d (for |exponent| ≤ 99)

+nnn
-nnn } Ew.d (for 99< |exponent|≤999)

E+n(1)n(2)...n(e)
E-n(1)n(2)...n(e) } Ew.dEe

The exponent field width specification is optional; if it is omitted, the value of e defaults to 2. If the exponent value is too large to be output in one of the above forms, an error occurs.

The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

The term w must be large enough to include: a minus sign when necessary (plus signs are optional), a zero, a decimal point, d digits, and an exponent. Therefore, w must be greater than or equal to d+7, or to d+e+5 if e is present.

Output Example:

Format	Internal Value	External Representation
E9.2	475867.222	△0.48E+06
E12.5	475867.222	△0.47587E+06
E12.3	0.00069	△△△0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E14.3E6	0.000123	△0.123E-000003

8.1.11 D Field Descriptor

The D field descriptor transfers real values in exponential form. It has the form

Dw.d

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

In an input statement, the D field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F and E field descriptors interpret and assign data in exactly the same way.

Input Example:

Format	External Field	Internal Value
BZ,D10.2	12345△△△△△	12345000.0D0
D10.2	△△123.45△△	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

In an output statement, the D field descriptor has the same effect as the E field descriptor, except that the D exponent field indicator is used in place of the E indicator.

Output Example:

Format	Internal Value	External Representation
D14.3	0.0363	△△△△△0.363D-01
D23.12	5413.87625793	△△△△△0.541387625793D+04
D9.6	1.2	*********

8.1.12 G Field Descriptor

The G field descriptor transfers real values in a form that, in effect, combines the F and E field descriptors. It has the form

Gw.d[Ee]

The corresponding I/O list element must be of real data type, or it must be either the real or the imaginary part of a complex data type.

In an input statement, the G field descriptor transfers w characters from the external field and assigns them as a real value to the corresponding I/O list element. The F, D, and E field descriptors interpret and assign data in exactly the same way.

In an output statement, the G field descriptor transfers the value of the corresponding I/O list element, rounded to d decimal positions and right justified, to an external field w characters long. The form in which the value is written is a function of the magnitude of the value, as described in Table 8-1.

Table 8-1
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m < 0.1	Ew.d[Ee]
0.1 ≤ m < 1.0	F(w-4).d, n(' ')
1.0 ≤ m < 10.0	F(w-4).(d-1), n(' ')
.	.
.	.
.	.
10**d-2≤m < 10**d-1	F(w-4).1, n(' ')
10**d-1≤m < 10**d	F(w-4).0, n(' ')
m ≥ 10**d	Ew.d[Ee]

The n(' ') field descriptor, which is, in effect, inserted by the G field descriptor for values within its range, specifies that four or e+2 spaces are to follow the numeric data representation.

The term w must be large enough to include: a minus sign when necessary (plus signs are optional); a decimal point; d digits to the right of the decimal point; and either a 4-character or e+2-character exponent. Therefore, w must be greater than or equal to d+7 or d+5+e.

Output Example:

Format	Internal Value	External Representation
G13.6	0.01234567	△0.123457E-01
G13.6	-0.12345678	-0.123457△△△△
G13.6	1.23456789	△△1.23457△△△△
G13.6	12.34567890	△△12.3457△△△△
G13.6	123.45678901	△△123.457△△△△
G13.6	-1234.56789012	△-1234.57△△△△
G13.6	12345.67890123	△△12345.7△△△△
G13.6	123456.78901234	△△123457.△△△△
G13.6	-1234567.89012345	-0.123457E+07

Compare the above example with the following example, which shows the same values output using an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	△△△△△0.012346
F13.6	-0.12345678	△△△△-0.123457
F13.6	1.23456789	△△△△△1.234568
F13.6	12.34567890	△△△△12.345679
F13.6	123.45678901	△△△123.456789
F13.6	-1234.56789012	△-1234.567890
F13.6	12345.67890123	△12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

8.1.13 L Field Descriptor

The L field descriptor transfers logical data. It has the form

Lw

The corresponding I/O list element must be of either integer or logical data type.

In an input statement, the L field descriptor transfers w characters from the external field. If the first nonblank characters of the field are T, t, .T, or .t, the value .TRUE. is assigned to the corresponding I/O list element; if the first nonblank characters are F, f, .F, or .f, the value .FALSE. is assigned. Any other value in the external field produces an error. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

In an output statement, the L field descriptor transfers either the letter T (if the value of the corresponding I/O list element is .TRUE.), or the letter F (if the value is .FALSE.) to an external field w characters long. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces.

Output Example:

Format	Internal Value	External Representation
L5	.TRUE.	△△△△T
L1	.FALSE.	F

8.1.14 A Field Descriptor

The A field descriptor transfers character or Hollerith values. It has the form

A[w]

The corresponding I/O list element can be of any data type. If it is of character data type, character data is transmitted. If it is of any other data type, Hollerith data is transmitted.

The value of w must be less than or equal to 32767.

In an input statement, the A field descriptor transfers w characters from the external record and assigns them to the corresponding I/O list element. The maximum number of characters that can be stored

depends on the size of the I/O list element. For character I/O list elements, the size is the length of the character variable, character substring reference, or character array element. For numeric I/O list elements, the size depends on the data type, as follows:

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*8(DOUBLE PRECISION)	8
REAL*16	16
COMPLEX	8 [1]
COMPLEX*16(DOUBLE COMPLEX)	16 [1]

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost characters are assigned to that element. The leftmost excess characters are ignored. If w is less than the number of characters that can be stored, w characters are assigned to the list element, left justified, and trailing spaces are added to fill the element.

Input Example:

Format	External Field	Internal Representation
A6	PAGE△#	# (CHARACTER*1)
A6	PAGE△#	E△# (CHARACTER*3)
A6	PAGE△#	PAGE△# (CHARACTER*6)
A6	PAGE△#	PAGE△#△△ (CHARACTER*8)
A6	PAGE△#	# (LOGICAL*1)
A6	PAGE△#	△# (INTEGER*2)
A6	PAGE△#	GE△# (REAL)
A6	PAGE△#	PAGE△#△△ (REAL*8)

In an output statement, the A field descriptor transfers the contents of the corresponding I/O list element to an external field w characters long. If w is greater than the list element size, the data appears in the field, right justified, with leading spaces. If w is less than the list element, only the leftmost w characters are transferred.

Output Example:

Format	Internal Value	External Representation
A5	OHMS	△OHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

If you omit w in an A field descriptor, a default value is supplied. If the I/O list element is of character data type, the default value is the length of the I/O list element. If the I/O list element is of numeric data type, the default value is the maximum number of characters that can be stored in a variable of that data type.

---

1. Because complex values are treated as pairs of real numbers, complex data editing requires two format codes. See Section 8.1.23.

8.1.15 H Field Descriptor

The H field descriptor transfers data between the external record and the H field descriptor itself. It has the form of a Hollerith constant

$$nHc_1c_2c_3 \ldots c_n$$

where:

n
is the number of characters to be transferred.

c
is an ASCII character.

In an input statement, the H field descriptor transfers n characters from the external field to the field descriptor. The first character appears immediately after the letter H. Any characters in the field descriptor before input are replaced by the input characters.

In an output statement, the H field descriptor transfers n characters following the letter H from the field descriptor to the external field.

8.1.15.1 **Character Constants** - You can use a character constant instead of an H field descriptor. Both types of format specifier function identically.

In a character constant, the apostrophe is written as two apostrophes. For example:

```
50   FORMAT ('TODAY''S△DATE△IS:△',I2,'/',I2,'/',I2)
```

A pair of apostrophes used this way is considered a single character.

8.1.16 X Edit Descriptor

The X edit descriptor is a positional specifier. It has the form

```
nX
```

The term n specifies how many character positions are to be passed over. The value of n must be greater than or equal to 1.

In an input statement, the X field descriptor specifies that the next n characters in the input record are to be skipped.

In an output statement, the X field descriptor tabs right n spaces; it does not write over anything already written. For example:

```
     WRITE (6,90) NPAGE
90   FORMAT ('1PAGE△NUMBER△',I2,16X,'GRAPHIC△ANALYSIS,△CONT.')
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn                GRAPHIC ANALYSIS, CONT.
```

The term nn is the current value of the variable NPAGE. The numeral 1 in the first H field descriptor is not printed, but is used to advance the printer paper to the top of a new page. Section 8.2 describes printer carriage control.

8.1.17 T Edit Descriptor

The T edit descriptor is a positional tabulation specifier. It has the form

```
Tn
```

The term n indicates the character position of the external record. The value of n must be greater than or equal to 1.

In an input statement, the T field descriptor positions the external record to its nth character position. For example, a READ statement inputs a record containing:

```
ABC△△△XYZ
```

This record is under the control of the FORMAT statement

```
10   FORMAT (T7,A3,T1,A3)
```

On execution, the READ statement would input the characters XYZ first, then the characters ABC.

In an output statement, the T field descriptor specifies that subsequent data transfer is to begin at the nth character position of the external record. The first position of a record to be printed is usually reserved for a carriage control character, which is not printed (see Section 8.2). For example:

```
     PRINT 25
25   FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

These statements would print the following line:

8.1.18 TL Edit Descriptor

The TL edit descriptor is a relative tabulation specifier. It has the form

```
TLn
```

The term n indicates that the next character to be transferred from or to a record is the nth character to the left of the current character. The value of n must be greater than or equal to 1. If the value of n is greater than or equal to the current character position, the first character in the record is specified.

8.1.19 TR Edit Descpriptor

The TR edit descriptor is also a relative tabulation specifier. It has the form

```
TRn
```

The term n indicates that the next character to be transferred from or to a record is the nth character to the right of the current character. The value of n must be greater than or equal to 1.

8.1.20 Q Edit Descriptor

The Q edit descriptor obtains the number of characters in the input record remaining to be transferred during a READ operation. It has the form

```
Q
```

The corresponding I/O list element must be of integer or logical data type.

For example:

```
          READ (4,1000) XRAY,KK,NCHRS,(ICHR(I),I=1,NCHRS)
1000      FORMAT (E15.7,I4,Q,80A1)
```

These input statements read two fields into the variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS and exactly that many characters are read into the array ICHR. By placing the Q descriptor first in the format specification, you can determine the actual length of the input record.

In an output statement, the Q edit descriptor has no effect except that the corresponding I/O list element is skipped.

8.1.21 Dollar Sign Descriptor

The dollar sign character ($) in a format specification modifies the carriage control specified by the first character of the record. In an output statement, the $ descriptor suppresses the carriage return if the first character of the record is a space or a plus sign. In an input statement, the $ descriptor is ignored. The $ descriptor is intended primarily for interactive I/O; it leaves the terminal print position at the end of the text (rather than returning it to the left margin) so that a typed response will follow the output on the same line.

Thus, the statements:

```
      TYPE 100
100   FORMAT (' ENTER RADIUS VALUE ',$)
      ACCEPT 200, RADIUS
200   FORMAT (F6.2)
```

produce a message on the terminal in the form

```
ENTER△RADIUS△VALUE
```

Your response (for example, 12.) can then go on the same line, as:

```
ENTER RADIUS VALUE 12.
```

8.1.22 Colon Descriptor

The colon character (:) in a format specification terminates format control if no more items are in the I/O list. The : descriptor has no effect if I/O list items remain. For example:

```
      PRINT 1,3
      PRINT 2,4
1     FORMAT(' I=',I2, ' J=',I2)
2     FORMAT(' K=',I2,:,' L=',I2)
```

These statements print the following two lines:

```
I=△3△J=
K=△4
```

Section 8.6 describes format control in detail.

8.1.23 Complex Data Editing

A complex value is an ordered pair of real values. Therefore, input or output of a complex value is governed by two real field descriptors, using any combination of the forms Fw.d, Ew.dEe, Dw.d, or Gw.dEe.

In an input statement, the two successive fields are read and assigned to a complex I/O list element as its real and imaginary parts, respectively.

Input Example:

Format	External Field	Internal Value
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 123456.789

In an output statement, the two parts of a complex value are transferred under the control of repeated or successive field descriptors. The two parts are transferred consecutively, without punctuation or spacing, unless the format specifier states otherwise.

Output Example:

Format	Internal Value	External Representation
2F8.5	2.3547188, 3.456732	△2.35472△3.45673
E9.2,'△,△',E5.3	47587.222, 56.123	△0.48E+06△,△*****

8.1.24 Scale Factor

The scale factor lets you alter, during input or output, the location of the decimal point in real values and in the two parts of complex values.

The scale factor has the form

nP

where:

n
is a signed or unsigned integer constant in the range -127 through +127. It specifies the number of positions, to the left or right, that the decimal point is to move.

A scale factor can appear anywhere in a format specification, but must precede the first field descriptor that is to be associated with it. For example:

nPFw.d nPEw.d nPDw.d nPGw.d

On input, the scale factor in any of the above field descriptors multiplies the data by 10**-n and assigns it to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. However, if the external field contains an explicit exponent, the scale factor has no effect.

Input Example:

Format	External Field	Internal Value
3PE10.5	△△△37.614△	.037614
3PE10.5	△△37.614E2	3761.4
-3PE10.5	△△△△37.614	37614.0

On output, the effect of the scale factor depends on the type of field descriptor associated with it. For the F field descriptor, the value of the I/O list element is multiplied by 10**n before transfer to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

For the E or D field descriptor, the basic real constant part of the I/O list element is multiplied by 10**n, and n is subtracted from the exponent. Thus, a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

Output Example:

Format	Internal Value	External Representation
1PE12.3	-270.139	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-0.03E+04

The effect of the scale factor for the G field descriptor is suspended if the magnitude of the data to be output is within the effective range of the descriptor, because the G field descriptor supplies its own scaling function. The G field descriptor functions as an E field descriptor if the magnitude of the data value is outside its range. In this case, the scale factor has the same effect as for the E field descriptor.

On input, and on output under F field descriptor control, a scale factor actually alters the magnitude of the data. On output, a scale factor under E, D, or G field descriptor control merely alters the form in which the data is transferred. In addition, on input, a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right; and, on output, the effect is the reverse.

If you do not specify a scale factor with a field descriptor, a default scale factor of 0 is assumed. Once you specify a scale factor, however, it applies to all subsequent real field descriptors in the same FORMAT statement, unless another scale factor appears. For example:

```
      DIMENSION A(6)
      DO 10 I = 1,6
10    A(I) = 25.
      TYPE 100,A
100   FORMAT(' ',F8.2,2PF8.2,F8.2)
```

produces:

```
  25.00 2500.00 2500.00  2500.00 2500.00 2500.00
```

If a second scale factor appears in the FORMAT statement, it takes control from the first scale factor.

Format reversion has no effect on the scale factor (see Section 8.6). A scale factor of 0 can only be reinstated by an explicit 0P specification.

8.1.25 Repeat Counts and Group Repeat Counts

You can apply the following field descriptors to a number of successive data fields by preceding the field descriptor with an unsigned integer constant specifying the number of repetitions: I, O, Z, F, E, D, G, L, and A. This constant is called a repeat count. For example, the following two statements are equivalent:

```
20    FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)

20    FORMAT (3E12.4,4I5)
```

Similarly, you can apply a group of field descriptors repeatedly to data fields by enclosing these field descriptors in parentheses and preceding them with an unsigned integer constant. The integer constant is called a group repeat count. For example, the following two statements are equivalent:

```
50    FORMAT (2I8,3(F8.3,E15.7),2(I5))

50    FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,2(I5))
                    1        2        3
```

An H or Q field descriptor, which could not otherwise be repeated, can be enclosed in parentheses and treated as a group repeat specification. Thus, it could be repeated a desired number of times.

If you do not specify a group repeat count, a default count of 1 is assumed.

Section 8.6 discusses how to use parentheses when the number of values to be formatted is greater than the number of format specifications.

8.1.26 Variable Format Expressions

You can use an expression in a FORMAT statement wherever you can use an integer (except as the specification of the number of characters in the H field) by enclosing it in angle brackets. For example:

```
FORMAT (I<J+1>)
```

This statement performs an I (integer) data transfer with a field width greater by 1 than the value of J at the time the format is scanned. The expression is reevaluated each time it is encountered in the normal format scan. If the expression is not of integer data type, it is converted to integer data type before being used. You can use any valid FORTRAN expression, including function calls and references to dummy arguments.

Figure 8-1 shows an example of a variable format expression.

The value of a variable format expression must obey the restrictions on magnitude applying to its use in the format, or an error occurs.

Variable format expressions are not permitted in run-time formats.

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./
C
      DO 10 I = 1,10
      WRITE (6,100) I
100   FORMAT(I<MAX(I,5)>)
10    CONTINUE
C

      DO 20 I = 1,5
      WRITE (6,101) (A(I),J=1,I)
101   FORMAT (<I>F10.<I-1>)
20    CONTINUE
      END
```

On execution, these statements produce the following output:

```
1
2
3
4
5
 6
  7
   8
    9
    10
    1.
   2.0        2.0
  3.00       3.00       3.00
 4.000      4.000      4.000      4.000
5.0000     5.0000     5.0000     5.0000     5.0000
```

Figure 8-1 Variable Format Expression Example

8.1.27 Default Field Descriptors

If you write the field descriptors I, O, Z, L, F, E, D, G, or A without specifying a field width value, default values for w, d, and e are supplied based on the data type of the I/O list element.

Table 8-2 lists the default values for w, d, and e.

Note that for the A field descriptor, the default is the actual length of the corresponding I/O list element.

8.2 CARRIAGE CONTROL

The first character of every record transferred to a printer is not printed. Instead, it is interpreted as a carriage control character (except when overridden by the OPEN statement keyword CARRIAGE CONTROL = 'LIST' or 'NONE'). The I/O system recognizes certain characters as carriage control characters. Table 8-3 lists these characters and their effects.

Table 8-2
Default Field Descriptor Values

Field Descriptor	List Element	w	d	e
I, O, Z	BYTE	7		
I, O, Z	INTEGER*2,LOGICAL*2	7		
I, O, Z	INTEGER*4,LOGICAL*4	12		
O, Z	REAL*4	12		
O, Z	REAL*8	23		
O, Z	REAL*16	44		
L	LOGICAL	2		
F, E, G, D	REAL, COMPLEX*8	15	7	2
F, E, G, D	REAL*8, COMPLEX*16	25	16	2
F, E, G, D	REAL*16	42	33	3
A	LOGICAL*1	1		
A	LOGICAL*2,INTEGER*2	2		
A	LOGICAL*4,INTEGER*4	4		
A	REAL*4, COMPLEX*8	4		
A	REAL*8, COMPLEX*16	8		
A	REAL*16	16		
A	CHARACTER*n	n		

Table 8-3
Carriage Control Characters

Character	Effect
△ (space)	Advances one line
0 (zero)	Advances two lines
1 (one)	Advances to top of next page
+ (plus)	Does not advance (allows overprinting)
$ (dollar sign)	Advances one line before printing and suppresses carriage return at the end of the record

Any character other than those listed in Table 8-3 is treated as a space and is deleted from the print line. Note that if you accidentally omit the carriage control character, the first character of the record is not printed.

8.3 FORMAT SPECIFICATION SEPARATORS

Field descriptors in a format specification are generally separated by commas. You can also use the slash (/) record terminator to separate field descriptors. A slash terminates input or output of the current record and initiates a new record. For example:

```
      WRITE (6,40) K,L,M,N,O,P
40    FORMAT (3O6/I6,2F8.4)
```

This statement is equivalent to:

```
      WRITE (6,40) K,L,M
40    FORMAT (3O6)
      WRITE (6,50) N,O,P
50    FORMAT (I6,2F8.4)
```

You can use multiple slashes to bypass input records or to output blank records. If n consecutive slashes appear between two field descriptors, (n-1) records are skipped on input, or (n-1) blank records are output. The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.

However, n slashes at the beginning or end of a format specification result in n skipped or blank records. This is because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example:

```
      WRITE (6,99)
99    FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

The above statements produce the following output:

```
Column 50, top of page
                ↓
                HEADING LINE
 (blank line)
                SUBHEADING LINE
 (blank line)
 (blank line)
```

8.4 EXTERNAL FIELD SEPARATORS

A field descriptor such as Fw.d specifies that an input statement is to read w characters from the external record. If the data field in the external record contains less than w characters, the input statement would read characters from the next data field in the external record, unless the short field is padded with leading zeros or spaces. When the field descriptor is numeric, you can avoid padding the input field by using a comma to terminate the field. The comma overrides the field descriptor's field width specification. This is called short field termination, and is particularly useful when you are entering data from a terminal keyboard. You can use it with the I, O, Z, F, E, D, G, and L field descriptors. For example:

```
      READ (5,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

The above statements read the following record:

```
1,-2,1.0,35
```

On execution, the following assignments occur:

```
I = 1

J = -2

A = 1.0

B = 0.35
```

Note that the physical end of the record also serves as a field terminator, and that the d part of a w.d specification is not affected by an external field separator.

You can use a comma to terminate only fields less than w characters long. If a comma follows a field of w characters or more, the comma is considered part of the next field.

Two successive commas, or a comma after a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.D0, 0.Q0, or .FALSE..

You cannot use a comma to terminate a field that is controlled by an A, H, or character constant field descriptor. However, if the record reaches its physical end before w characters are read, short field termination occurs and the characters that were input are assigned successfully. Trailing spaces are appended to fill the corresponding I/O list element or the field descriptor.

8.5 RUN-TIME FORMAT

You can store format specifications in character variables, character arrays, character array elements, character substrings, character expressions, numeric arrays, or numeric array elements. Such a format specification is called a run-time format, and can be constructed or altered during program execution.

A run-time format in an array has the same form as a FORMAT statement, without the word FORMAT and the statement label. The opening and closing parentheses are required. Variable format expressions are not permitted in run-time formats.

In the following example, the DATA statement assigns a left parenthesis to the character array element FORCHR(0), and assigns a right parenthesis and three field descriptors to four character variables for later use. Next, the proper field descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of the array TABLE. A right parenthesis is then added to the format specification just before the WRITE statement uses it. Thus, the format specification changes with each iteration of the DO loop.

```
      SUBROUTINE PRINT(TABLE)
      REAL TABLE(10,5)
      CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG,FMED,FSML
      DATA FORCHR(0),RPAR/'(',')'/
      DATA FBIG,FMED,FSML/'F8.2,','F9.4,','F9.6,'/
      DO 20 I=1,10
          DO 18 J=1,5
              IF (TABLE(I,J) .GE. 100.) THEN
                  FORCHR(J)=FBIG
              ELSE IF (TABLE(I,J) .GT. 0.1) THEN
                  FORCHR(J)=FMED
              ELSE
                  FORCHR(J)=FSML
              END IF
18        CONTINUE
          FORCHR(5)(5:5)=RPAR
          WRITE (6,FORCHR) (TABLE(I,J), J=1,5)
20    CONTINUE
      END
```

NOTE

Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a READ statement to read data into the format itself, that data is not copied back into the original array. Thus, it will not be available subsequently for using that array as a run-time format specification.

8.6 FORMAT CONTROL INTERACTION WITH INPUT/OUTPUT LISTS

Format control begins with execution of a formatted I/O statement. The action taken by format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the format specification. Both the I/O list and the format specification are interpreted from left to right, except when repeat counts and implied DO lists are specified.

If the I/O statement contains an I/O list, you must specify at least one I, O, Z, F, E, D, G, L, A, or Q field descriptor in the format specification. An error occurs if these conditions are not met.

On execution, a formatted input statement reads one record from the specified unit and initiates format control. Thereafter, additional records can be read as indicated by the format specification. Format control requires that a new record be input when a slash occurs in the format specification, or when the last closing parenthesis of the format specification is reached and I/O list elements remain to be filled. Any remaining characters in the current record are discarded when the new record is read.

On execution, a formatted output statement transmits a record to the specified unit as format control terminates. Records can also be written during format control if a slash appears in the format specification or if the last closing parenthesis is reached and more I/O list elements remain to be transferred.

The I, O, Z, F, E, D, G, L, A, and Q field descriptors each correspond to one element in the I/O list. No list element corresponds to an H, X, P, T, TL, TR, SP, SS, S, BN, BZ, $, :, or character constant field descriptor. In H and character constant field descriptors, data transfer occurs directly between the external record and the format specification.

When an I/O list element is to be transferred, format field descriptors are processed -- beginning with the current format item -- until a descriptor is found that corresponds to an I/O list element. The I/O list element is then transferred under control of the field descriptor.

Format execution continues until one of the following is encountered: an element-transferring field descriptor; a : edit descriptor; or the end of the format. These also terminate format execution when no I/O list elements are to be transferred.

When the last closing parenthesis of the format specification is reached, format control determines whether more I/O list elements are to be processed. If not, format control terminates. However, if additional list elements remain, part or all of the format specification is reused in a process called format reversion.

In format reversion, the current record is terminated, a new one is initiated, and format control reverts to the group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification. If the format does not contain a group repeat specification, format control returns to the initial opening parenthesis of the format specification. Format control continues from that point.

8.7 SUMMARY OF RULES FOR FORMAT STATEMENTS

The following sections summarize the rules for constructing and using the format specifications and their components, and for constructing external fields and records. Table 8-4 summarizes the FORMAT codes.

8.7.1 General Rules

1. A FORMAT statement must always be labeled.

2. In a field descriptor such as rIw[.m] or nX, the terms r, w, m, and n must be unsigned integer constants greater than zero, or variable format expressions whose values are greater than zero. (They cannot be names assigned to constants in PARAMETER statements.) You can omit the repeat count and field width specification.

3. In a field descriptor such as Fw.d, the term d must be an unsigned integer constant or variable format expression. You must specify d in F, E, D, and G field descriptors even if it is zero. The decimal point is also required. You must either specify both w and d, or omit them both. In a field descriptor such as Ew.dEe, the term e must also be an unsigned integer constant.

4. In a field descriptor such as nHc1c2 ... c_n, exactly n characters must follow the H format code. You can use any printing ASCII character in this field descriptor.

5. In a scale factor of the form nP, n must be an integer constant or variable format expression in the range -127 through 127 inclusive. The scale factor affects the F, E, D, and G field descriptors only. Once you specify a scale factor, it applies to all subsequent real field descriptors in that format specification until another scale factor appears. You must explicitly specify 0P to reinstate a scale factor of zero. Format reversion does not affect the scale factor.

6. No repeat count is permitted in BN, BZ, S, SS, SP, H, Q, X, T, TR, TL, $, :, or character constant field descriptors unless these descriptors are enclosed in parentheses and treated as a group repeat specification.

7. If the associated I/O statement contains an I/O list, the format specification must contain at least one field descriptor. This descriptor must be I, O, Z, F, E, D, G, L, A, or Q.

8. A format specification in a character variable, character substring reference, character array element, character array, character expression, numeric array, or numeric array element must be constructed in the same way as a format specification in a FORMAT statement, including the opening and closing parentheses.

9. If a character-constant format includes apostrophes, those apostrophes must be represented by double apostrophes.

8.7.2 Input Rules

1. A minus sign must precede a negative value in an external input field; a plus sign is optional before a positive value.

2. On input, an external field under I field descriptor control must be an integer constant. It cannot contain a decimal point or an exponent. An external field under O field descriptor control must contain only the numerals 0 through 7. An external field input under Z field descriptor control must contain only the numerals 0 through 9 and the letters A(a) through F(f). An external field under O or Z field descriptor control must not contain a sign, a decimal point, or an exponent. You cannot use octal and hexadecimal constants in the form '777'O or 'AF9'X in external records.

3. On input, an external field under F, E, D, or G field descriptor control must be an integer constant or a real constant. It can contain a decimal point and/or an E(e), D(d), or Q(q) exponent field.

4. If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real field descriptor.

5. If an external field contains an exponent, the scale factor (if any) of the corresponding field descriptor is inoperative for the conversion of that field.

6. The field width specification must be large enough to accommodate both the numeric character string of the external field and any other characters that are allowed (algebraic sign, decimal point, and/or exponent).

7. A comma is the only character you can use as an external field separator. It terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It also designates null (zero-length) fields.

8.7.3 Output Rules

1. A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

2. The field width specification (w) must be large enough to accommodate all characters that the data transfer can generate, including an algebraic sign, decimal point, and exponent. For example, the field width specification in an E field descriptor should be large enough to contain d+7 or d+e+5 characters.

3. The first character of a record output to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a space, 0,1, $, or +. Any other character is treated as a space and deleted from the record.

Table 8-4
Summary of FORMAT Codes

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values
BN	BN	Specifies that embedded and trailing blanks in a numeric input field are to be ignored
BZ	BZ	Specifies that embedded and trailing blanks in a numeric input field are to be treated as zeros
D	Dw.d	Transfers real values (D exponent field indicator)
E	Ew.d[Ee]	Transfers real values (E exponent field indicator)
F	Fw.d	Transfers real values
G	Gw.d[Ee]	Transfers real values: on input, acts like F code; on ouput, acts like E code or F code, depending on the magnitude of the value

(continued on next page)

Table 8-4 (Cont.)
Summary of FORMAT Codes

Code	Form	Effect
H	nHc...c	Transfers data between the H field decriptor and an external record
I	Iw[.m]	Tranfers decimal integer values
L	Lw	Transfers logical data: on input, transfers characters; on output, transfers T or F
O	Ow[.m]	Transfers octal values
Q	Q	Obtains the number of characters remaining to be transferred in an input record
S	S	Reinvokes optional plus characters in numeric output fields: counters the action of SP and SS
SP	SP	Writes plus characters that would otherwise be optional into numeric output fields
SS	SS	Suppresses optional plus characters in numeric output fields
T	Tn	Positional tabulation specifier
TL	TLn	Relative tabulation specifier (left)
TR	TRn	Relative tabulation specifier (right)
X	nX	Specifies that n characters are to be skipped
Z	Zw[.m]	Transfers hexadecimal values
$	$	Suppresses carriage return during interactive I/O
:	:	Terminates format control if the I/O list is exhausted

CHAPTER 9

AUXILIARY INPUT/OUTPUT STATEMENTS

The auxiliary input/output statements perform file management functions. These statements are:

- OPEN -- associates FORTRAN logical units with files. OPEN establishes a connection between a logical unit and a file or device, and declares the attributes required for read and write operations.
- CLOSE -- terminates the connection between a logical unit and a file or device.
- INQUIRE -- inquires about specified properties of a file or logical unit.
- REWIND and BACKSPACE -- perform file-positioning functions.
- ENDFILE -- writes a special form of record that causes an end-of-file condition (and END= transfer) when an input statement reads the record.
- DELETE -- deletes a record from a file.
- UNLOCK -- unlocks a currently accessed record, permitting access by other programs.

OPEN

9.1 OPEN STATEMENT

An OPEN statement either connects an existing file to a logical unit, or creates a new file and connects it to a logical unit. In addition, OPEN can specify file attributes that control file creation and/or subsequent processing.

The OPEN statement has the form

```
OPEN(par[,par]...)
```

where:

par
is a keyword specification in one of the following forms:

```
key
key = value
```

where:

key
is a keyword, as described below.

value
depends on the keyword, as described below.

Keywords are divided into several categories based on function:

- Keywords that identify the unit and file:

UNIT	- logical unit number to be used
FILE or NAME	- file name specification for the file
STATUS or TYPE	- file existence status at OPEN
DISPOSE	- file existence status after CLOSE

- Keywords that describe the file processing to be performed:

ACCESS	- FORTRAN access method to be used
ORGANIZATION	- logical file structure
READONLY	- write protection

- Keywords that describe the records in the file:

BLOCKSIZE	- physical block size
CARRIAGECONTROL	- printer control type
FORM	- type of FORTRAN record formatting
RECL or RECORDSIZE	- logical record length
RECORDTYPE	- logical record format
BLANK	- blank interpretation for numeric input
KEY	- keys for keyed access

- Keywords that describe file storage allocation when a file is created:

INITIALSIZE	- initial file allocation
EXTENDSIZE	- file allocation increment size

- Keywords that provide additional capability for direct-access I/O:

ASSOCIATEVARIABLE	- the next record number value
MAXREC	- maximum direct access record number

- Optional keywords that provide improved performance or special capabilities. These options are generally transparent to I/O processing:

BUFFERCOUNT	- number of I/O buffers to use
NOSPANBLOCKS	- records are not to be split across physical blocks
SHARED	- other programs can simultaneously access the file
USEROPEN	- user program option to provide additional OPEN capability

ERR - statement to which control is transferred if an error occurs during execution of the OPEN statement

IOSTAT - status value that indicates whether an error condition exists

Table 9-1 lists the values accepted for each keyword.

Table 9-1
OPEN Statement Keyword Values

Keyword	Values*	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'	Access method	'SEQUENTIAL'
ASSOCIATEVARIABLE	v	Next direct access record	
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	e	Physical block size	System default
BUFFERCOUNT	e	Number of I/O buffers	System default
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	'FORTRAN' (formatted) 'NONE' (unformatted)
DISPOSE DISP	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'	File disposition at close	'KEEP'
ERR	s	Error transfer label	
EXTENDSIZE	e	File allocation increment	Volume or system default
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on ACCESS keyword
FILE NAME	c	File name specification	

Key: v is an integer variable name
e is a numeric expression
s is a statement label
c is a character expression, numeric array name, numeric variable name, or numeric array element name

(continued on next page)

Table 9-1 (Cont.)
OPEN Statement Keyword Values

Keyword	Values*	Function	Default
INITIALSIZE	e	File allocation	
IOSTAT	v	Input/output status	
KEY	e1:e2 [:INTEGER] [:CHARACTER]	Key field definitions	
MAXREC	e	Direct access record limit	
NOSPANBLOCKS	-	Records do not span blocks	
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'	File structure	'SEQUENTIAL'
READONLY	-	Write protection	
RECL RECORDSIZE	e	Record length	As specified at file creation
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'	Record structure	Depends on ORGANIZATION, ACCESS, and FORM keywords
SHARED	-	File sharing allowed	
STATUS TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
UNIT	e	Logical unit number	
USEROPEN	p	User program option	

Key: e is a numeric expression
v is an integer variable name
e1 is the first byte position of a key
e2 is the last byte position of a key
p is an external function

You can specify character values at run time by substituting a general character expression for a keyword value in the OPEN statement. The character value may contain trailing spaces, but it must not contain either leading or embedded spaces. For example:

```
CHARACTER*7 DELETE
     .
     .
     .
OPEN (UNIT=1, STATUS='NEW', DISP='SUBMIT'//DELETE)
```

Keyword specifications can appear in any order. In most cases, they are optional; default values are provided in their absence. If the logical unit specifier is the first parameter in the list, the keyword identifier [UNIT=] is optional.

The following examples illustrate four uses of the OPEN statement:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

This statement creates a new sequential formatted file on unit 1 with the default file name FOR001.DAT.

```
OPEN (UNIT=3, STATUS='SCRATCH', ACCESS='DIRECT',
1     INITIALSIZE=50, RECORDSIZE=64)
```

This statement creates a 50-block direct access file for temporary storage. The file is deleted at program termination.

```
OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', BLOCKSIZE=8192,
1     STATUS='NEW', ERR=14, RECL=1024, RECORDTYPE='FIXED')
```

This statement creates a file on magnetic tape with a large block size for efficient processing.

```
OPEN (UNIT=I, FILE='MTA0:MYDATA.DAT', READONLY, STATUS='OLD',
1     RECL=1024, RECORDTYPE='FIXED', BLOCKSIZE=8192)
```

This statement opens the file created in the previous example for input.

```
CHARACTER*40 FILENAME
     .
     .
     .
OPEN (UNIT=1, FILE=FILENAME, STATUS='OLD')
```

This statement opens an existing file, using the name specified by the character variable FILENAME.

Sections 9.1.1 through 9.1.27 describe in detail the parameters represented by the various keywords. As used in these sections, a numeric expression can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

9.1.1 ACCESS Keyword

The ACCESS parameter has the form

```
ACCESS = acc
```

where:

acc
is a character expression having a value equal to 'DIRECT', 'SEQUENTIAL', 'KEYED', or 'APPEND'.

ACCESS specifies whether the file is keyed, direct, or sequential access. If you specify 'DIRECT', the file is accessed directly. If you specify 'SEQUENTIAL', the file is accessed sequentially. If you specify 'KEYED', the file is accessed by a specified key. 'APPEND' implies sequential access and positioning after the last record of the file. The default is 'SEQUENTIAL'.

9.1.2 ASSOCIATEVARIABLE Keyword

The ASSOCIATEVARIABLE parameter has the form

```
ASSOCIATEVARIABLE = asv
```

where:

asv
is an integer variable.

ASSOCIATEVARIABLE specifies the integer variable (asv) that, after each direct access I/O operation, contains the record number of the next sequential record in the file; asv must not be a dummy argument. This specifier is ignored for an other than direct-access file.

9.1.3 BLANK Keyword

The BLANK parameter has the form

```
BLANK = blnk
```

where:

blnk
is a character expression having a value equal to either 'NULL' or 'ZERO'

BLANK specifies either that all blanks in a numeric input field are to be ignored (except if the field is comprised of all blanks, in which case it is treated as zero), or that all blanks other than leading blanks are to be treated as zeros. The default value is 'NULL'.

If the /NOF77 compiler command qualifier is specified, the default value is 'ZERO'.

9.1.4 BLOCKSIZE Keyword

This keyword has the format

```
BLOCKSIZE = bks
```

where:

bks
is a numeric expression.

BLOCKSIZE specifies the physical I/O transfer size (in bytes) for the file. The default is the system default for the device. See the VAX-11 FORTRAN User's Guide for more information.

9.1.5 BUFFERCOUNT Keyword

The BUFFERCOUNT parameter has the form

```
BUFFERCOUNT = bc
```

where:

bc
is a numeric expression.

BUFFERCOUNT specifies the number of buffers to be associated with the logical unit for multibuffered I/O. The BLOCKSIZE keyword determines the size of each buffer. If you do not specify BUFFERCOUNT, or if you specify zero, the system default is assumed.

9.1.6 CARRIAGECONTROL Keyword

The CARRIAGECONTROL parameter has the form

```
CARRIAGECONTROL = cc
```

where:

cc
is a character expression having a value equal to 'FORTRAN', 'LIST', or 'NONE'.

CARRIAGECONTROL determines the kind of carriage control processing to be used when printing a file. The default for formatted files is 'FORTRAN'; for unformatted files, the default is 'NONE'. 'FORTRAN' specifies normal FORTRAN interpretation of the first character, 'LIST' specifies single spacing between records, and 'NONE' specifies no implied carriage control.

9.1.7 DISPOSE Keyword

The DISPOSE (or DISP) parameter has the form

```
DISPOSE = dis
DISP = dis
```

where:

dis
is a character expression having a value equal to 'KEEP', 'SAVE', 'DELETE', 'PRINT', 'SUBMIT', 'PRINT/DELETE', or 'SUBMIT/DELETE'.

DISPOSE determines the disposition of the file connected to the unit when the unit is closed. If you specify 'KEEP' or 'SAVE', the file is retained after the unit is closed; this is the default value. If you specify 'DELETE', the file is deleted. If you specify 'PRINT', the file is submitted to the system line printer spooler and is not deleted; it is deleted if you specify 'PRINT/DELETE'. If you specify 'SUBMIT,' the file is submitted to the batch job queue and is not deleted; it is deleted if you specify 'SUBMIT/DELETE'. A read-only file cannot be deleted. A scratch file cannot be saved, printed, or submitted.

9.1.8 ERR Keyword

The ERR parameter has the form

ERR= s

where:

s
is the label of an executable statement.

ERR transfers control to the executable statement specified by s when an error occurs. ERR applies only to the OPEN statement in which it is specified, and not to subsequent I/O operations on the unit. If an error occurs, no file is opened or created.

9.1.9 EXTENDSIZE Keyword

The EXTENDSIZE parameter has the form

EXTENDSIZE = es

where:

es
is a numeric expression.

EXTENDSIZE specifies the number of blocks by which to extend a disk file when additional file storage is allocated. If you do not specify EXTENDSIZE, or if you specify zero, the system default for the device is used.

9.1.10 FILE Keyword

The FILE parameter has the form

FILE = fln

where:

fln
is a character expression, numeric array name, numeric variable name, or numeric array element name.

FILE specifies the name of the file to be connected to the unit. The name can be any file specification accepted by the operating system. The VAX-11 FORTRAN User's Guide describes default file name conventions.

If the file name is stored in a numeric variable, numeric array, or numeric array element, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character variable, array, or array element, it must not contain a zero byte.

9.1.11 FORM Keyword

The FORM parameter has the form

FORM = ft

where:

ft
is a character expression having a value equal to 'FORMATTED' or 'UNFORMATTED'.

FORM specifies whether the file being opened is to be read and written using formatted or unformatted READ or WRITE statements. For sequential access files, 'FORMATTED' is the default. For direct-access and keyed-access files, 'UNFORMATTED' is the default.

9.1.12 INITIALSIZE Keyword

The INITIALSIZE parameter has the form

INITIALSIZE = insz

where:

insz
is a numeric expression.

INITIALSIZE specifies the number of blocks in the initial allocation of space for a new file on a disk. If you do not specify INITIALSIZE, or if you specify zero, no initial allocation is made.

9.1.13 IOSTAT Keyword

The IOSTAT parameter has the form

```
IOSTAT = ios
```

where:

ios
is an integer variable or integer array element.

IOSTAT is an input/output status specifier. It causes ios to be defined as zero if no error condition exists, or as a positive integer if an error condition exists. VAX-11 FORTRAN input/output status values are described in the VAX-11 FORTRAN User's Guide. IOSTAT applies only to the OPEN statement in which it appears and not to subsequent I/O operations on the logical unit that is opened; however, IOSTAT can be used in subsequent I/O statements to perform a similar function (see Chapter 7).

9.1.14 KEY Keyword

The KEY parameter has the form

```
KEY=(kspec[,kspec]...)
```

where:

kspec
has the form

```
el:e2[:dt]
```

where:

el
is the first byte position of the key.

e2
is the last byte position of the key.

dt
is the data type of the key: either INTEGER or CHARACTER.

KEY defines the access keys for records in an indexed file. The key starts at position el in the record and has a length of e2-el+1. The values of el and e2 must be such that:

```
1 .LE. (el) .LE. (e2) .LE. record-length.
1 .LE. (e2-el+1) .LE. 255
```

If the key type is INTEGER, the key length must be either 2 or 4. There may be up to 255 key specifications in a list, but there must be at least one. The first key specification defines the primary key, the second defines the first alternate key, and so forth. The default data type is CHARACTER. The position of a key specification in the list determines a key's key-of-reference number. This number is used in any subsequent I/O statement to specify the same key. The primary key is key-of-reference number 0, the first alternate key is key-of-reference number 1, and so forth.

The key fields and key-of-reference numbers are permanent attributes of an indexed file and are established when the file is created. The KEY parameter must be specified when a file is created, but need not be specified when an existing file is opened. When an existing file is opened, key definitions and key-of-reference numbers are obtained from the file itself. If the KEY parameter is specified for an existing file, it must agree with the established attributes of the file.

9.1.15 MAXREC Keyword

The MAXREC parameter has the form

```
MAXREC = mr
```

where:

mr
is a numeric expression.

MAXREC specifies the maximum number of records permitted in a direct-access file. The default is no maximum number of records. This specifier applies only to direct-access files.

9.1.16 NAME Keyword

A nonstandard synonym for FILE. See Section 9.1.10.

9.1.17 NOSPANBLOCKS Keyword

The NOSPANBLOCKS parameter has the form

```
NOSPANBLOCKS
```

NOSPANBLOCKS specifies that records are not to cross disk block boundaries. If any record exceeds the size of a physical block, an error occurs.

9.1.18 ORGANIZATION Keyword

The ORGANIZATION parameter has the form

```
ORGANIZATION = org
```

where:

org
is a character expression whose value is equal to 'SEQUENTIAL', 'RELATIVE', or 'INDEXED'.

ORGANIZATION specifies the internal organization of the file. When you create a file, the default is 'SEQUENTIAL'. When you access an existing file, the default is the organization of that file. If you specify ORGANIZATION for an existing file, org must have the same value as that of the existing file.

See the VAX-11 FORTRAN User's Guide for more information on internal file organization.

9.1.19 READONLY Keyword

The READONLY parameter has the form

```
READONLY
```

READONLY specifies that an existing file can be read, but prohibits writing to that file.

9.1.20 RECL Keyword

The RECL parameter has the form

```
RECL = rl
```

where:

rl
is a numeric expression.

RECL specifies the logical record length. If the file contains fixed-length records, RECL specifies the size of each record. If the file contains variable-length records, RECL specifies the maximum length for any record. If the records are formatted, the length is the number of bytes, whereas if the records are unformatted, the length is the number of longwords. If the file exists and rl does not agree with the actual length of the record, an error occurs. If you omit this specifier for old files, the actual record length specified when the file was created is assumed. You must specify RECL when you create files with fixed-length records, or with relative or indexed organization.

9.1.21 RECORDSIZE Keyword

A nonstandard synonym for RECL. See Section 9.1.20

9.1.22 RECORDTYPE Keyword

The RECORDTYPE parameter has the form

```
RECORDTYPE = typ
```

where:

typ
is a character expression whose value is equal to 'FIXED', 'VARIABLE', or 'SEGMENTED'.

RECORDTYPE specifies whether the file has fixed-length records, variable-length records, or segmented records. When you create a file, the defaults are:

File Type	Default Record Type
Relative or indexed files	'FIXED'
Direct-access sequential files	'FIXED'
Formatted sequential-access files	'VARIABLE'
Unformatted sequential-access files	'SEGMENTED'

A segmented record consists of one or more variable-length records. Using segmented records allows a FORTRAN logical record to span several physical records. Only sequential access, unformatted files with sequential organization can use segmented records. You cannot specify 'SEGMENTED' for any other file type.

If you do not specify RECORDTYPE when accessing an existing file, the record type of the file is used. An exception to this is sequential access, unformatted files with sequential organization; these files have a default of 'SEGMENTED'.

If you specify RECORDTYPE, typ must match the record type of the existing file.

In fixed-length record files, if an output statement does not specify a full record, the record is filled with spaces (for a formatted file) or zeros (for an unformatted file).

9.1.23 SHARED Keyword

The SHARED parameter has the form

```
SHARED
```

SHARED specifies that the file is to be opened for shared access by more than one program executing simultaneously.

See the VAX-11 FORTRAN User's Guide for additional information on this keyword.

9.1.24 STATUS Keyword

The STATUS parameter has the form

```
STATUS = sta
```

where:

sta
is a character expression whose value is equal to 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'.

STATUS specifies the status of the file to be opened. If you specify 'OLD', the file must already exist. If you specify 'NEW', a new file is created. If you specify 'SCRATCH', a new file is created and it is deleted when the file is closed. If you specify 'UNKNOWN', the processor will first try 'OLD'; if the file is not found, the processor will use 'NEW', thereby creating a new file. The default is 'UNKNOWN'.

If the /NOF77 compiler command qualifier is specified, the default value is 'NEW'.

NOTE

The STATUS parameter is also used in CLOSE statements to specify the status of a file after the file is closed; however, the values it uses are different from those used in OPEN statements.

9.1.25 **TYPE Keyword**

A nonstandard synonym for STATUS. See Section 9.1.24.

9.1.26 **UNIT Keyword**

The UNIT parameter has the form

[UNIT=] u

where:

u
is a numeric expression.

UNIT specifies the logical unit to which a file is to be connected. The unit specification must appear in the list. The optional character string UNIT= can be omitted only when the UNIT parameter occupies the first position in the list.

The logical unit may already be connected to a file when an OPEN statement is executed. If this file is not the same as the one to be opened, the OPEN statement executes as if a CLOSE statement had executed just before it. If the file to be opened is already connected to the unit, or if the FILE= specifier is not included in the OPEN statement, only the BLANK= specifier may have a value different from the one currently in effect. The position of the file is unaffected.

9.1.27 USEROPEN Keyword

The USEROPEN parameter has the form

```
USEROPEN = p
```

where:

p
is an external function name.

The USEROPEN keyword specifies a user-written external function that controls the opening of the file. Knowledgeable users can employ additional features of the operating system that are not directly available from FORTRAN, while retaining the convenience of writing programs in FORTRAN. See the VAX-11 FORTRAN User's Guide for more information on USEROPEN.

CLOSE

9.2 CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit. It has the form

```
                    STATUS
CLOSE ([UNIT=]u [,DISPOSE   = p] [,ERR=s][,IOSTAT=ios])
                    DISP
```

where:

u
is a logical unit number.

p
is a character expression that determines the disposition of the file. Its values are 'KEEP', 'SAVE', 'DELETE', 'PRINT', 'SUBMIT,' 'SUBMIT/DELETE', and 'PRINT/DELETE'.

s
is the label of an executable statement.

ios
is an integer variable or integer array element.

The CLOSE statement parameters can occur in any order. The keyword UNIT= is optional only if the unit specifier is the first parameter in the list.

If you specify either 'SAVE' or 'KEEP', the file is retained after the unit is closed. If you specify 'DELETE', the file is deleted. If you specify 'PRINT', the file is submitted to the line printer spooler; it is deleted if you specify 'PRINT/DELETE'. If you specify 'SUBMIT', the file is submitted to the batch job queue; it is deleted if you specify 'SUBMIT/DELETE'. For scratch files, the default is 'DELETE'; for all other files, the default is 'KEEP'. The disposition specified in a CLOSE statement supersedes the disposition specified in the OPEN statement, except that a file opened as a scratch file cannot be saved, printed, or submitted, nor can a file opened for read-only access be deleted.

For example:

```
CLOSE(UNIT=1, STATUS='PRINT')
```

This statement closes the file on unit 1 and submits the file for printing.

```
CLOSE(UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file on unit J and deletes it.

INQUIRE

9.3 INQUIRE STATEMENT

The INQUIRE statement inquires about specified properties of a file or of a logical unit on which a file might be opened. The INQUIRE statement has two forms, one for inquiring by file and the other for inquiring by unit:

```
INQUIRE ( FILE = fi, flist)
INQUIRE ( [UNIT =]u, ulist)
```

where:

fi
is a character expression, numeric array name, numeric variable name, or numeric array element name whose value specifies the name of the file to be inquired about.

flist
is a property-specifier list in which any one specifier appears only once.

u
is the number of the logical unit to be inquired about. The unit need not exist, nor need it be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

ulist
is a property-specifier list in which any one specifier appears only once.

FILE=fi and UNIT=u can appear anyplace in the property-specifier list; however, if UNIT= is omitted, u must be the first property in the list.

An INQUIRE statement may be executed before, during, or after the connection of a file to a unit; the values assigned by the statement are those that are current at the time of execution of the INQUIRE statement.

The following specifiers may be used in either form of the INQUIRE statement:

9.3.1 ACCESS Specifier

The ACCESS specifier has the form

```
ACCESS = acc
```

where:

acc
is a character variable, array element, or substring reference.

Acc is assigned the value SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, and KEYED if the file is connected for keyed access. If there is no connection, acc is UNKNOWN.

9.3.2 BLANK Specifier

The BLANK specifier has the form

```
BLANK = blk
```

where:

blk
is a character variable, array element, or substring reference.

Blk is assigned the value NULL if null blank control is in effect for a file connected for formatted I/O, and the value ZERO if zero blank control is in effect. If there is no connection, or if the connection is not for formatted I/O, blk is UNKNOWN.

9.3.3 CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier has the form

```
CARRIAGECONTROL = cc
```

where:

cc
is a character variable, array element, or substring reference.

Cc is assigned the value FORTRAN if the file has the FORTRAN carriage control attribute, LIST if the file has the implied carriage control attribute, NONE if the file has no carriage control attribute, and UNKNOWN if no other value applies.

9.3.4 DIRECT Specifier

The DIRECT specifier has the form

```
DIRECT = dir
```

where:

dir
is a character variable, array element, or substring reference.

Dir is assigned the value YES if DIRECT is an allowed access method for the file, NO if DIRECT is not an allowed access method, and UNKNOWN if the processor is unable to determine whether DIRECT is an allowed access method.

9.3.5 ERR Specifier

The ERR specifier has the form

```
ERR = s
```

where:

s
is the label of an executable statement.

ERR is a control specifier rather than a property specifier. If an error occurs during execution of the INQUIRE statement, control is transferred to the statement whose label is s.

9.3.6 EXIST Specifier

The EXIST specifier has the form

```
EXIST = ex
```

where:

ex
is a logical variable or logical array element.

Ex is assigned the value true if the specified file or unit exists, and the value false if the specified file or unit does not exist.

9.3.7 FORM Specifier

The FORM specifier has the form

```
FORM = fm
```

where:

fm
is a character variable, array element, or substring reference.

Fm is assigned the value FORMATTED if the file is connected for formatted I/O, and UNFORMATTED if the file is connected for unformatted I/O. If there is no connection, fm is UNKNOWN.

9.3.8 FORMATTED Specifier

The FORMATTED specifier has the form

```
FORMATTED = fmd
```

where:

fmd
is a character variable, array element, or substring reference.

Fmd is assigned the value YES if formatted is an allowed form for the file, NO if formatted is not an allowed form, and UNKNOWN if the processor is unable to determine whether formatted is an allowed form.

9.3.9 IOSTAT Specifier

The IOSTAT specifier has the form

```
IOSTAT = ios
```

where:

ios
is an integer variable or integer array element.

IOSTAT is a control specifier rather than a property specifier. Ios is assigned a processor-dependent positive integer value if an error occurs during execution of the INQUIRE statement, and the value zero if there is no error condition.

9.3.10 KEYED Specifier

The KEYED specifier has the form

```
KEYED = kyd
```

where:

kyd
is a character variable, array element, or substring reference.

Kyd is assigned the value YES if KEYED is an allowed access method for the file (that is, the file is indexed), NO if KEYED is not an allowed access method, and UNKNOWN if the processor is unable to determine whether KEYED is an allowed access method.

9.3.11 NAME Specifier

The NAME specifier has the form

```
NAME = nme
```

where:

nme
is a character variable, array element, or substring reference.

Nme is assigned the name of the file being inquired about. If the file does not have a name, name is not defined.

The value assigned to nme is not necessarily identical to the value specified with FILE=. For example, the value that the processor returns may be qualified by a directory name or version number. However, the value that is assigned is always valid for use with FILE= in an OPEN statement.

NOTE

FILE and NAME are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

9.3.12 **NAMED Specifier**

The NAMED specifier has the form

NAMED = nmd

where:

nmd
is a logical variable or logical array element.

Nmd is assigned the value true if the specified file has a name, and the value false if it does not have a name.

9.3.13 **NEXTREC Specifier**

The NEXTREC specifier has the form

NEXTREC = nr

where:

nr
is an integer variable or integer array element.

Nr is assigned an integer value which is 1 more than the number of the last record read or written on the specified direct-access file. If no records have been read or written, the value of nr is 1. If the file is not connected for direct access, or if the position is indeterminate because of an error condition, nr is zero.

9.3.14 **NUMBER Specifier**

The NUMBER specifier has the form

NUMBER = num

where:

num
is an integer variable or integer array element.

Num is assigned the number of the logical unit currently connected to the specified file. If there is no logical unit connected to the file, num is not defined.

9.3.15 OPENED Specifier

The OPENED specifier has the form

```
OPENED = od
```

where:

od
is a logical variable or logical array element.

Od is assigned the value true if the specified file is opened on a unit, or if the specified unit is opened; it is assigned the value false if the file or unit is not open.

9.3.16 ORGANIZATION Specifier

The ORGANIZATION specifier has the form

```
ORGANIZATION= org
```

where:

org
is a character variable, array element, or substring reference.

Org is assigned the value SEQUENTIAL if the file is a sequential file, RELATIVE if the file is a relative file, and INDEXED if the file is an indexed file. If the processor is unable to determine the organization, org is assigned the value UNKNOWN.

9.3.17 RECL Specifier

The RECL specifier has the form

```
RECL = rcl
```

where:

rcl
is an integer variable or integer array element.

If the file (or unit) is opened, rcl is the maximum record length allowed; if not opened, rcl is the longest record in the file. If a specified file does not exist, rcl is zero. Rcl is expressed in bytes if the file is opened for formatted input/output, and in longwords if the file is unformatted.

9.3.18 RECORDTYPE Specifier

The RECORDTYPE specifier has the form

```
RECORDTYPE = rtype
```

where:

rtype
is a character variable, array element, or substring reference.

Rtype is assigned the value FIXED if the file has fixed-length records, VARIABLE if the file has variable-length records, and SEGMENTED if the file is connected for unformatted sequential input/output using segmented records. If the processor cannot determine the recordtype, rtype is assigned the value UNKNOWN.

9.3.19 SEQUENTIAL Specifier

The SEQUENTIAL specifier has the form

```
SEQUENTIAL = seq
```

where:

seq
is a character variable, array element, or substring reference.

Seq is assigned the value YES if SEQUENTIAL is an allowed access method for the specified file, NO if SEQUENTIAL is not an allowed access method, and UNKNOWN if the processor cannot determine whether SEQUENTIAL is an allowed access method.

9.3.20 UNFORMATTED Specifier

The UNFORMATTED specifier has the form

```
UNFORMATTED = unf
```

where:

unf
is a character variable, array element, or substring reference.

Unf is assigned the value YES if unformatted is an allowed form for the file, NO if unformatted is not an allowed form for the file, and UNKNOWN if the processor is unable to determine whether unformatted is an allowed form for the file.

REWIND

9.4 REWIND STATEMENT

The REWIND statement repositions a sequential file currently open for sequential or append access to the beginning of the file. It has the forms

```
REWIND ([UNIT=]u[,ERR=s][,IOSTAT=ios])

REWIND u
```

where:

u
is a logical unit number.

s
is the label of the executable statement to which control is to be transferred if an error occurs.

ios
is an integer variable or integer array element that is assigned a positive integer if an error occurs, and zero if no error occurs.

The unit number must refer to a file on disk or magnetic tape.

For example:

```
REWIND 3
```

This statement repositions logical unit 3 to the beginning of the currently open file.

You must not issue a REWIND statement for a file that is open for direct or keyed access.

BACKSPACE

9.5 BACKSPACE STATEMENT

The BACKSPACE statement repositions a sequential file currently open for sequential access to the beginning of the preceding record. When the next I/O statement for the unit is executed, this preceding record is available for processing.

The BACKSPACE statement has the forms

```
BACKSPACE ([UNIT=]u[,ERR=s][,IOSTAT=ios])

BACKSPACE u
```

where:

u
is a logical unit number.

s
is the label of the executable statement to which control is to be transferred if an error occurs.

ios
is an integer variable or integer array element that is defined as a positive integer if an error occurs, and zero if no error occurs.

The unit number must refer to an open file on disk or magnetic tape. For example:

```
BACKSPACE 4
```

This statement repositions the open file on logical unit 4 to the beginning of the preceding record.

You must not issue a BACKSPACE statement for a file that is open for direct, keyed, or append access.

ENDFILE

9.6 ENDFILE STATEMENT

The ENDFILE statement writes an end-file record to the specified unit. It has the form

```
ENDFILE ([UNIT=]u[,ERR=s][,IOSTAT=ios])

ENDFILE u
```

where:

u
is a logical unit number.

s
is the label of the executable statement to which control is to be transferred if an error occurs.

ios
is an integer variable or integer array element that is defined as a positive integer if an error occurs, and zero if no error occurs.

An end-file record can be written only to sequentially accessed sequential files containing variable-length records.

For example:

```
ENDFILE 2
```

This statement outputs an end-file record to logical unit 2.

You must not issue an ENDFILE statement for a file that is open for direct or keyed access.

9.7 DELETE STATEMENT

The DELETE statement deletes records from relative and indexed files. It has the forms

```
DELETE (u[,ERR=s][,IOSTAT=ios])

DELETE (u,REC=r[,ERR=s][,IOSTAT=ios])

DELETE (u'r[,ERR=s][,IOSTAT=ios])
```

where:

u
is the number of the logical unit containing the record to be deleted.

r
is the positional number of the record to be deleted.

s
is the label of an executable statement to which control is to be transferred if an error condition occurs.

ios
is an integer variable or integer array element that is defined as a positive integer if an error occurs, and zero if no error occurs.

The first form of the DELETE statement above is a current-record delete. This form of the statement deletes the current record, which is the last record to be accessed by a READ statement on the specified logical unit.

The second and third forms of the DELETE statement above are direct-access deletes. These forms of the statement are used only to directly access relative files; they delete the record specified by the number r.

The DELETE Statement logically removes the appropriate record from the specified file, and then frees the position formerly occupied by that record so a new record may be written into it. After a direct-access delete, any associated variable is set to the next record number.

The following examples demonstrate the use of the DELETE statement.

```
DELETE (10,REC=5)
```

In this example, the fifth record in the file connected to logical unit 10 is deleted from the file.

```
DELETE (11)
```

In this example, the current record is deleted from the file connected to logical unit 11.

UNLOCK

9.8 UNLOCK STATEMENT

The UNLOCK statement unlocks a record in a relative or indexed file, locked by a previous READ, without performing any other I/O operations. It has the forms

```
UNLOCK ([unit=]u[,ERR=s][,IOSTAT=ios])
UNLOCK u
```

where:

u
is the number of a logical unit.

s
is the label of the executable statement to which control is to be transferred if an error occurs.

ios
is an integer variable or integer array element that is defined as a positive integer if an error occurs, and zero if no error occurs.

The UNLOCK statement frees a previously locked record on the specified logical unit. If no record is locked, the operation has no effect.

APPENDIX A

ADDITIONAL LANGUAGE ELEMENTS

For the purpose of facilitating compatibility with other versions of FORTRAN, VAX-11 FORTRAN includes the statements ENCODE, DECODE, DEFINE FILE, and FIND, and offers alternative syntax for the PARAMETER statement and octal constants. These language elements are particularly useful in transporting older FORTRAN programs to VAX-11, but should be avoided in new FORTRAN programs for VAX-11, or where portability to other FORTRAN-77 implementations is important.

Section A.6 describes the interpretation of the EXTERNAL statement that applies when the /NOF77 compiler command qualifier is used. The ANSI FORTRAN-77 interpretation is incompatible with the previous standard and with previous Digital FORTRAN implementations.

ENCODE
DECODE

A.1 THE ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements transfer data between variables or arrays in internal storage, and translate that data from internal to character form, and vice versa, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE and READ statements.

The ENCODE and DECODE statements have the forms

```
ENCODE(c,f,b [,IOSTAT=ios][,ERR=s])[list]
DECODE(c,f,b [,IOSTAT=ios][,ERR=s])[list]
```

where:

c
: is an integer expression. (In the ENCODE statement, c is the number of characters (bytes) to be translated to character form. In the DECODE statement, c is the number of characters to be translated to internal form)

f
: is a format identifier. (If more than one record is specified, an error occurs.)

b
: is the name of an array, array element, variable, or character substring reference. (In the ENCODE statement, b receives the characters after translation to external form. In the DECODE statement, b contains the characters to be translated to internal form.)

ios
is an integer variable or integer array element that is defined as a positive integer if an error occurs, and as a zero if no error occurs.

s
is the label of an executable statement.

list
is an I/O list. (In the ENCODE statement, the I/O list contains the data to be translated to character form. In the DECODE statement, the list receives the data after translation to internal form.)

The ENCODE statement translates the list elements to character form according to the format specifier, and stores the characters in b, as does a WRITE statement. If fewer than c characters are transmitted, the remaining character positions are filled with spaces.

The DECODE statement translates the character data in b to internal (binary) form according to the format specifier, and stores the elements in the list, as does a READ statement.

If b is an array, its elements are processed in the order of subscript progression.

The number of characters that the ENCODE or DECODE statement can process depends on the data type of b in that statement. For example, an INTEGER*2 array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array. A character variable or character array element can contain characters equal in number to its length. A character array can contain characters equal in number to the length of each element multiplied by the number of elements.

The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

An example of the ENCODE and DECODE statements follows:

```
      DIMENSION K(3)
      CHARACTER*12 A, B
      DATA A /'123456789012'/
      DECODE (12,100,A) K
100   FORMAT (3I4)
      ENCODE (12,100,B) K(3), K(2), K(1)
```

The DECODE statement translates the 12 characters in A to integer form (specified by statement 100), and stores them in array K, as follows:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B as follows:

```
B = '901256781234'
```

DEFINE FILE

A.2 DEFINE FILE STATEMENT

The DEFINE FILE statement describes unformatted direct access files that are associated with a logical unit number. The OPEN statement performs the same function, and its use is preferred. The DEFINE FILE statement establishes the size and structure of the direct access file.

The DEFINE FILE statement has the form

```
DEFINE FILE u (m,n,U,asv) [,u(m,n,U,asv)] ...
```

where:

u
: is an integer constant or variable that specifies the logical unit number.

m
: is an integer constant or variable that specifies the number of records in the file.

n
: is an integer constant or variable that specifies the length of each record in 16-bit words (2 bytes).

U
: specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv
: is an integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to v; asv must not be a dummy argument.

The DEFINE FILE statement specifies that a file containing m fixed-length records of n 16-bit words each exists, or is to exist, on the specified logical unit. The records in the file are numbered sequentially from 1 through m.

A DEFINE FILE statement must be executed before the first direct-access I/O statement referring to the specified file, even though the DEFINE FILE statement does not itself open the file. The file is actually opened when the first direct-access I/O statement for the unit is executed. If this I/O statement is a WRITE, a new direct-access file is created. If it is a READ or FIND, an existing file is opened -- unless, of course, the specified file does not exist, in which case an error occurs.

The DEFINE FILE statement also establishes the integer variable asv as the associated variable of a file. At the end of each direct-access I/O operation, the FORTRAN I/O system places in asv the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or FIND statement), direct-access I/O statements can perform sequential processing on the file. They do this by using the associated variable of the file as the record number specifier.

For example:

```
DEFINE FILE 3 (1000,48,U,NREC)
```

This statement specifies that Logical Unit 3 is to be connected to a file of 1000 fixed-length records; each record is 48 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

FIND

A.3 FIND STATEMENT

The FIND statement positions a direct-access file on a specified unit to a particular record and sets the associated variable of the file to that record number. No data transfer takes place.

The FIND statement has the forms

```
FIND (u'r [,ERR=s][,IOSTAT=ios])
FIND ([UNIT=]u, REC=r [,ERR=s][,IOSTAT=ios])
```

where:

u
: is a logical unit number.

r
: is the direct-access record number.

s
: is the label of the executable statement to which control is to be transferred if no error occurs.

ios
: is an integer variable or integer array element that is defined as a positive integer if an error occurs, and as a zero if no error occurs.

The unit number must refer to a direct-access file.

The record number cannot be less than 1 or greater than the number of records defined for the file.

For example:

```
FIND (1'1)
```

This statement positions logical unit 1 to the first record of the file; the file's associated variable is set to 1.

```
FIND (4'INDX)
```

This statement positions the file to the record identified by the content of INDX; the file's associated variable is set to the value of INDX.

PARAMETER

A.4 PARAMETER STATEMENT

As does the PARAMETER statement discussed in Section 5.10, this statement assigns a symbolic name to a constant. However, it differs from the PARAMETER statement discussed in Section 5.10 in that its list is not bounded with parentheses, and the form of the constant rather than implicit or explicit typing of the symbolic name determines the data type of the variable.

The PARAMETER statement has the form

```
PARAMETER  p=c  [,p=c] ...
```

where:

p
: is a symbolic name.

c
: is a constant, the symbolic name of a constant, or a compile-time constant expression.

Each symbolic name (p) becomes a constant and is defined as the value of the constant or constant expression (c).

Compile-time constant expressions are defined in Section 5.10

Once a symbolic name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the symbolic name.

The symbolic name of a constant cannot appear as part of another constant, but it can appear as a real or imaginary part of a complex constant.

You can use a symbolic name in a PARAMETER statement only to identify the symbolic name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The symbolic name of a constant assumes the data type of its corresponding constant expression. The initial letter of the constant's name does not affect its data type. You cannot specify the data type of a parameter constant in a type declaration statement.

For example:

```
PARAMETER  PI=3.1415927, DPI=3.141592653589793238D0
PARAMETER  PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER  FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

A.5 OCTAL NOTATION FOR INTEGER CONSTANTS

Octal forms of integer constants are provided for compatibility with PDP-11 FORTRAN.

The octal form of an integer constant is:

```
"nn
```

where:

nn
is a string of digits in the range 0 to 7.

Examples of valid and invalid octal integer constants are:

Valid	Invalid	
"107	"108	(contains a digit outside the allowed range)
"177777	"1377.	(contains a decimal point)
	"17777"	(contains a trailing quotation mark)

Note that these octal forms are not the same as the typeless octal constants discussed in Section 2.3.7. Integer constants in octal form have integer data type and are treated as integers.

EXTERNAL

A.6 /NOF77 INTERPRETATION OF THE EXTERNAL STATEMENT

The /NOF77 interpretation of the EXTERNAL statement combines the function of the INTRINSIC statement with that of the EXTERNAL statement discussed in Section 5.7. It is available only if the /NOF77 compiler command qualifier is present.

The /NOF77 EXTERNAL statement allows the programmer to use subprograms as arguments to other subprograms.

The subprograms to be used as arguments can be either user-supplied procedures or FORTRAN library functions.

The /NOF77 EXTERNAL statement has the form

```
EXTERNAL [*]v [,[*]v]...
```

where:

v
is the symbolic name of a subprogram, or the name of a dummy argument associated with the symbolic name of a subprogram.

*
specifies that a user-supplied function is to be used instead of a FORTRAN library function having the same name. See Section 6.3 for information on FORTRAN library functions.

The /NOF77 EXTERNAL statement declares that each symbolic name in its list is an external procedure name. Such a name can then be used as an actual argument to a subprogram, which can use the corresponding dummy argument in a function reference or CALL statement.

Note, however, that a complete function reference used as an argument -- SQRT(B) in CALL SUBR(A,SQRT(B),C), for example -- represents a value, not a subprogram name. It is not, therefore, defined in an EXTERNAL statement (as would be the incomplete reference SQRT).

An example of the /NOF77 EXTERNAL statement follows:

Main Program

```
EXTERNAL SIN,COS,*TAN,SINDEG
   .
   .
CALL TRIG (ANGLE,SIN,SINE)
   .
   .
CALL TRIG (ANGLE,COS,COSINE)
   .
   .
CALL TRIG (ANGLE,TAN,TANGNT)
   .
   .
CALL TRIG (ANGLED,SINDEG,SINE)
   .
   .
   .
```

Subprograms

```
SUBROUTINE TRIG (X,F,Y)
Y = F(X)
RETURN
END


FUNCTION TAN (X)
TAN = SIN(X) / COS(X)
RETURN
END

FUNCTION SINDEG(X)
SINDEG = SIN (X*3.14159/180)
RETURN
END
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)
```

The functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN library. The function TAN is also supplied in the library. But the asterisk in the EXTERNAL statement specifies that the user-supplied function be used, instead of the library function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

APPENDIX B

CHARACTER SETS

B.1 FORTRAN CHARACTER SET

The FORTRAN character set consists of:

1. The letters A through Z and a through z
2. The numerals 0 through 9
3. The following special characters:

Character	Name	Character	Name
△	Space or tab	'	Apostrophe
=	Equal sign	"	Quotation mark
+	Plus sign	$	Dollar sign
-	Minus sign	_	Underline
*	Asterisk	!	Exclamation point
/	Slash	:	Colon
(	Left parenthesis	<	Left angle bracket
)	Right parenthesis	>	Right angle bracket
,	Comma	%	Percent sign
.	Period	&	Ampersand

Other printing characters can appear in a FORTRAN statement only as part of a Hollerith or character constant. Any printing character can appear in a comment. Printing characters are characters whose ASCII codes are in the range 20 through 7D. See Table B-1.

B.2 ASCII CHARACTER SET

Table B-1 is a table representing the ASCII character set. At the top of the table are hexadecimal digits (0 to 7), and to the left of the table are hexadecimal digits (0 to F). To determine the hexadecimal value of an ASCII character, use the hexadecimal digit that corresponds to the row in the "units" position, and use the hexadecimal digit that corresponds to the column in the "16's" position. For example, the value of the character representing the equal sign is 3D.

Table B-1
ASCII Character Set

	Columns							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tab	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space	DEL	Delete

B.3 RADIX-50 CONSTANTS AND CHARACTER SET

Radix-50 is a special character data representation in which up to 3 characters can be encoded and packed into 16 bits. The Radix-50 character set is a subset of the ASCII character set, and is provided for compatibility with PDP-11 FORTRAN.

The Radix-50 characters and their corresponding code values are:

Character	ASCII Octal Equivalent	Radix-50 Value (Octal)
Space	40	0
A - Z	101 - 132	1 - 32
$	44	33
.	56	34
(Unassigned)		35
0 - 9	60 - 71	36 - 47

Radix-50 values are stored, up to 3 characters per word, by packing them into single numeric values according to the formula:

```
((i * 50 + j) * 50 + k)
```

where:

i, j, and k
represent the code values of 3 Radix-50 characters.

Thus, the maximum Radix-50 value is:

```
47*50*50 + 47*50 + 47 = 174777
```

A Radix-50 constant has the form

$nRc_1c_2...c_n$

where:

n
is an unsigned, nonzero integer constant that states the number of characters to follow.

c
is a character from the Radix-50 character set.

The maximum number of characters is 12. The character count must include any spaces that appear in the character string (the space character is a valid Radix-50 character). You can use Radix-50 constants only in DATA statements.

Examples of valid and invalid Radix-50 constants are:

Valid	Invalid
4RABCD	4RDK0: (colon is not a Radix-50 character)
6R△TO△△△	

When a Radix-50 constant is assigned to a numeric variable or array element, the number of bytes that can be assigned depends on the data type of the component (see Table 2-2). If the Radix-50 constant contains fewer bytes than the length of the component, ASCII null characters (zero bytes) are appended on the right. If the constant contains more bytes than the length of the component, the rightmost characters are not used.

APPENDIX C

FORTRAN LANGUAGE SUMMARY

C.1 EXPRESSION OPERATORS

The following lists the expression operators in each data type in order of descending precedence:

Data Type	Operator	Operation	Operates upon:
Arithmetic	**	Exponentiation	Arithmetic or logical expressions
	*,/	Multiplication, division	
	+,-	Addition, subtraction, unary plus and minus	
Character	//	Concatenation	Character expressions
Relational	.GT.	Greater than	Arithmetic, logical, or character expressions (all relational operators have equal precedence)
	.GE.	Greater than or equal to	
	.LT.	Less than	
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	
Logical	.NOT.	.NOT.A is true if and only if A is false	Logical or integer expressions
	.AND.	A.AND.B is true if and only if A and B are both true	

Data Type	Operator	Operation	Operates upon:
Logical (Cont.)	.OR.	A.OR.B is true if either A or B or both are true	
	.EQV.	A.EQV.B is true if and only if A and B are both true or A and B are both false	.EQV., .NEQV., and .XOR. have equal priority
	.XOR.	A.XOR.B is true if and only if A is true and B is false or B is true and A is false	
	.NEQV.	Same as .XOR..	

C.2 STATEMENTS

The following summarizes the statements available in the VAX-11 FORTRAN language, including the general form of each statement. The statements are listed alphabetically for ease of reference. The "Manual Section" column indicates the section of this manual that describes each statement in detail.

Form	Effect	Manual Section
ACCEPT	See READ	7.7
Arithmetic/Logical/Character Assignment		3.1, 3.2, 3.3

v=e

v — is a variable name, an array element name, or a character substring name.

e — is an expression.

Assigns the value of the arithmetic, logical, or character expression to the variable.

ASSIGN s TO v — 3.4

s — is the label of a FORMAT statement or an executable statement.

v — is an integer variable name.

Associates the statement label s with the integer variable v for later use as a format specifier or in an assigned GO TO statement.

Form	Effect	Manual Section

BACKSPACE ([UNIT=]u[,IOSTAT=ios][,ERR=s]) 9.5
BACKSPACE u

u is a logical unit specifier.

ios is an integer variable or integer array element.

s is the label of an executable statement

Backspaces the currently open file on logical unit u one record.

BLOCK DATA [nam] 5.12

nam is a symbolic name.

Specifies the subprogram that follows as a BLOCK DATA subprogram.

CALL f[([a][,[a]]...)] 4.6, 6.2

f is a subprogram name or entry point.

a is an expression, an array name, a procedure name, or an alternate return specifier. An alternate return specifier is *s or &s, where s is the label of an executable statement.

Calls the subroutine subprogram with the name specified by f, passing the actual arguments a to replace the dummy arguments in the subroutine definition.

CLOSE ([UNIT=]u[,p][,IOSTAT=ios][,ERR=s]) 9.2

p is one the following parameters:

```
STATUS     'SAVE'
DISPOSE =  'KEEP'
DISP       'DELETE'
           'PRINT'
           'SUBMIT'
           'PRINT/DELETE'
           'SUBMIT/DELETE'
```

u is a logical unit specifier.

s is the label of an executable statement.

ios is an integer variable or integer array element.

Closes the specified file.

Form	Effect	Manual Section
COMMON [/[cb]/] nlist [[,]/[cb]/nlist]...		5.4
	cb is a common block name.	
	nlist is a list of one or more variable names, array names, or array declarators separated by commas.	
	Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name.	
CONTINUE		4.5
	Causes no processing.	
DATA nlist/clist/[[,] nlist/clist/]...		5.9
	nlist is a list of one or more variable names, array names, array element names, character substring names, or implied DO lists, separated by commas. Subscript expressions and substring expressions must be constant.	
	clist is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.	
	Initially stores elements of clist in the corresponding elements of nlist.	
DECODE (c,f,b[,IOSTAT=ios][,ERR=s])[list]		A.1
	c is an integer expression.	
	f is a format specifier.	
	b is a variable name, array name, array element name, or character substring name.	
	ios is an input/output status specifier.	
	s is a label of an executable statement.	
	list is an I/O list.	
	Reads c characters from buffer b and assigns values to the elements in the list converted according to format specification f.	

Form	Effect	Manual Section

DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]... A.2

u — is a logical unit specifier.

m — is a constant or variable.

n — is a constant or variable.

U — specifies unformatted.

v — is an integer variable name.

Defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed-length records in the file, n is the length in 16-bit words of a single record, U is a fixed argument, and v is the associated variable.

DELETE ([UNIT=]u[,REC=r][,IOSTAT=ios][,ERR=s]) 9.7
DELETE (u'r[,IOSTAT=ios][,ERR=s])

u — is a logical unit specifier.

r — is a relative record number specifier.

ios — is an input/output specifier.

s — is the label of an executable statement.

Deletes records from relative or indexed files, where u is the logical unit connected to the file, r is the number of the record in a relative file, ios is an input/output status specifier, and s is the label of the statement to which control is to be transferred if an error occurs.

DIMENSION a(d)[,a(d)]... 5.3

a(d) — An array declarator.

Specifies storage space requirements for arrays.

Form	Effect	Manual Section

DO [s [,]] v = e1,e2[,e3] 4.3.1

s is the label of an executable statement.

v is a variable name.

ei,e2,e3 are numeric expressions.

Executes the DO loop by performing the following steps:

1. Evaluates cnt= INT((e2-e1+e3)/e3)
2. Sets v=e1
3. If cnt is less than or equal to zero, does not execute the loop
4. If cnt is greater than zero, then
   a. Executes the statements in the body of the loop
   b. Evaluates v=v+e3
   c. Decrements the loop count (cnt=cnt-1). If cnt is greater than zero, repeats the loop

DO [s[,]]WHILE(e) 4.3.2

s is a statement label.

e is a logical expression.

Similar to the DO statement, but executes as long as the logical expression contained in the statement continues to be true, instead of for a specified number of iterations.

ELSE 4.2.3

Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false. See IF THEN.

Form	Effect	Manual Section
ELSE IF (e) THEN		4.2.3
e	is a logical expression.	
	Defines a block of statements to be executed if logical expressions in previous IF THEN and ELSE IF THEN statements have values of false, and the logical expression e has a value of true. See IF THEN.	
ENCODE (c,f,b[,IOSTAT=ios][,ERR=s])[list]		A.1
c	is an integer expression.	
f	is a format specifier.	
b	is a variable name, array name, array element name, or substring name.	
ios	is an input/output specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Writes c characters into buffer b, which contains the values of the elements of the list, converted according to format specification f.	
END		4.10
	Delimits a program unit.	
END DO		4.4
	May be used in place of a labeled statement to delimit the body of a DO loop.	
ENDFILE ([UNIT=]u[,IOSTAT=ios][,ERR=s]) ENDFILE u		9.6
u	is a logical unit specifier.	
ios	is an input/output specifier.	
s	is the label of an executable statement.	
	Writes an end-file record on logical unit u.	

Form	Effect	Manual Section
END IF	Terminates block IF construct. See IF THEN.	4.2.3
ENTRY nam [([p[,p]...])]		6.2.4
nam	is a subprogram name.	
p	is a symbolic name or an alternate return specifier(*).	
	Defines an alternate entry point within a subroutine or function subprogram.	
EQUIVALENCE (nlist)[,(nlist)]...		5.5
nlist	is a list of two or more variable names, array names, array element names, or character substring names separated by commas. Subscript expressions and substring expressions must be compile-time constant expressions.	
	Assigns each of the names in nlist the same storage location.	
EXTERNAL v[,v]...		5.7, A.6
v	is a subprogram name.	
	Defines the names specified as user-defined subprograms.	
FIND ([UNIT=]u,REC=r[,IOSTAT=ios][ERR=s]) FIND (u'r[,IOSTAT=ios][,ERR=s])		A.3
u	is a logical unit specifier.	
r	is an integer expression.	
ios	is an input/output specifier.	
s	is the label of an executable statement.	
	Positions the file on logical unit u to record r and sets the associated variable to record number r.	

Form	Effect	Manual Section
FORMAT (field specification,...)	Describes the format in which one or more records are to be transmitted; a statement label must be present.	8.1 - 8.7
[typ] FUNCTION nam[*n][([p[,p]...])]	**typ** is a data type specifier. **nam** is a symbolic name. ***n** is a data type length specifier. p is a symbolic name. Begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.	6.2.2
GO TO s	s is a label of an executable statement. Transfers control to statement number s.	4.1.1
GO TO (slist)[,] e	**slist** is a list of one or more statement labels separated by commas. e is an integer expression. Transfers control to the statement specified by the value of e (if e=1, control transfers to the first statement label; if e=2, control transfers to the second statement label, and so forth). If e is less than 1 or greater than the number of statement labels present, no transfer takes place.	4.1.2
GO TO v [[,](slist)]	v is an integer variable name. **slist** is a list of one or more statement labels separated by commas. Transfers control to the statement most recently associated with v by an ASSIGN statement.	4.1.3

Form	Effect	Manual Section

IF (e) s1,s2,s3 — 4.2.1

e — is an expression.

s1,s2,s3 are labels of executable statements.

Transfers control to statement si depending on the value of e (if e is less than zero, control transfers to s1; if e equals zero, control transfers to s2; if e is greater than zero, control transfers to s3).

IF (e) st — 4.2.2

e — is an expression.

st is any executable statement except a DO, END DO, END, block IF, or logical IF.

Executes the statement if the logical expression has a value of true.

IF (e1) THEN — 4.2.3

block

ELSE IF (e2) THEN

block

ELSE

block

END IF

e1,e2 are logical expressions.

block is a series of zero or more FORTRAN statements.

Defines blocks of statements and conditionally executes them. If the logical expression in the IF THEN statement has a value of true, the first block is executed and control transfers to the first executable statement after the END IF statement.

Form	Effect	Manual Section

If the logical expression has a value of false, the process is repeated for the next ELSE IF THEN statement. If all logical expressions have values of false, the ELSE block is executed. If there is no ELSE block, control transfers to the next executable statement following END IF.

IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]... 5.1

typ is a data type specifier.

a is either a single letter, or two letters in alphabetical order separated by a hyphen (that is, X-Y).

The element a represents a single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that data type.

INCLUDE 'file specification[/[NO]LIST]' 1.5

'file specification'
is a character constant.

Includes the source statements in the compilation from the file specified.

INQUIRE (par[,par]...) 9.3

par is a keyword specification having the form

key=variable

key is a keyword as described below.

value depends on the keyword.

Keyword	Values
inputs	
FILE	fin
UNIT	e

Form	Effect	Manual Section

INQUIRE (par[,par]...) (Cont.)

outputs

ACCESS	cv
BLANK	cv
CARRIAGECONTROL	cv
DIRECT	cv
ERR	s
EXIST	lv
FORM	cv
FORMATTED	cv
IOSTAT	v
KEYED	cv
NAMED	lv
NEXTREC	v
NUMBER	v
OPENED	lv
ORGANIZATION	cv
RECL	v
RECORDTYPE	cv
SEQUENTIAL	cv
UNFORMATTED	cv

e is a numeric expression.

fin is a character expression.

v is an integer variable or integer array element.

lv is a logical variable or array element.

cv is a character variable, array element, or substring reference.

s is the label of an executable statement.

Furnishes information on specified characteristics of a file or of a logical unit and its connections to a file.

INTRINSIC fun [,fun]... 5.8

fun is an intrinsic function name.

Identifies symbolic names as representing intrinsic functions and allows those names to be used as actual arguments.

Form	Effect	Manual Section
OPEN(par[,par]...)		9.1

par is a keyword specification in one of the following forms:

```
key
key = value
```

key is a keyword, as described below.

value depends on the keyword.

Keyword	Values
ACCESS	'SEQUENTIAL' 'DIRECT' 'KEYED' 'APPEND'
ASSOCIATEVARIABLE	v
BLOCKSIZE	e
BLANK	'NULL' 'ZERO'
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
DISP	(same as DISPOSE)
DISPOSE	'KEEP' or 'SAVE' 'PRINT' 'DELETE' 'SUBMIT' 'SUBMIT/DELETE' 'PRINT/DELETE'
ERR	s
EXTENDSIZE	e
FILE	c
FORM	'FORMATTED' 'UNFORMATTED'
INITIALSIZE	e
IOSTAT	v
KEY	keyspec
MAXREC	e
NAME	(Same as File)
NOSPANBLOCKS	-
ORGANIZATION	'SEQUENTIAL' 'RELATIVE' 'INDEXED'
READONLY	-
RECL	e
RECORDSIZE	(same as RECL)
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'
SHARED	-
STATUS	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'
TYPE	(same as STATUS)
UNIT	e
USEROPEN	p

Form	Effect	Manual Section
	c is a character expression, numeric array name, numeric variable name, numeric array element name, or Hollerith constant.	
	e is a numeric expression.	
	p is a program unit name.	
	s is a statement label.	
	v is an integer variable name.	
	keyspec is (el:e2[:type]),	
	where:	
	el is the beginning byte of the key field	
	e2 is the ending byte of the key field	
	type is either INTEGER or CHARACTER	
	Opens a file on the specified logical unit according to the parameters specified by the keywords.	
PARAMETER (p=c [,p=c]...)		5.10, A.4
	p is a symbolic name.	
	c is a constant or compile-time constant expression.	
	Defines a symbolic name for a constant.	
PAUSE [disp]		4.8
	disp is a decimal digit string containing 1 to 5 digits or a character constant	
	Suspends program execution and prints the display, if one is specified.	
PRINT	See WRITE	7.8
PROGRAM nam		5.11
	nam A symbolic name.	
	Specifies a name for the main program.	

Form	Effect	Manual Section
READ ([UNIT=]u,[FMT=]f[,IOSTAT=ios][,END=s][,ERR=s])[list]		7.4.1
READ f[,list]		7.4.1
ACCEPT f[,list]		7.8

u — is a logical unit specifier.

f — is a format specifier.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Reads one or more logical records from unit u and assigns values to the elements in the list. The records are converted according to the format specifier (f).

READ ([UNIT=]u,[FMT=]*[,IOSTAT=ios][,END=s][,ERR=s])list		7.4.1.2
READ *,[list]		7.4.1.2
ACCEPT *,[list]		7.7

u — is a logical unit specifier.

* — denotes list-directed formatting.

s — is a label of an executable statement.

list — is an I/O list.

Reads one or more logical records from unit u and assigns values to the elements in the list. The records are converted according to the data type of the list element.

READ([UNIT=]u[,IOSTAT=ios][,END=s][,ERR=s])[list]		7.4.1.3

u — is a logical unit specifier.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Reads one unformatted record from unit u and assigns values to the elements in the list.

Form	Effect	Manual Section

READ ([UNIT]=u,[FMT=]f,REC=r[,iostat=ios][,ERR=s])[list] 7.4.2.1

READ (u'r,[FMT=]f[,IOSTAT=ios][,ERR=s])[list] 7.4.2.1

u is a logical unit specifier.

r is a relative record number specifier.

f is a format specifier.

ios is an input/output specifier.

s is a label of an executable statement.

list is an I/O list

Reads record r from unit u and assigns values to the elements in the list. The record is converted according to f.

READ([UNIT=]u,REC=r[,IOSTAT=ios][,ERR=s])[list] 7.4.2.2

READ(u'r[,ERR=s])[list] 7.4.2.2

u is a logical unit specifier.

r is a relative record number specifier.

ios is an input/output specifier.

s is a label of an executable statement.

list is an I/O list.

Reads record r from unit u and assigns values to the elements in the list.

READ ([UNIT=]u, [FMT=]f,keyspec[,KEYID=kn][,IOSTAT=ios] [,ERR=s])[list] 7.4.3

READ ([UNIT=]u, keyspec[,KEYID=kn][,IOSTAT=ios] [,ERR=s])[list]

u is a logical unit specifier.

f is a format specifier.

keyspec is a key specifier (see Section 7.2.1.5).

kn is a key-of-reference specifier.

s is the label of an executable statement.

list is an I/O list.

Reads one or more logical records, specified by key value, and assigns values to the elements in the list.

Form	Effect	Manual Section

READ ([UNIT=]c,FMT=f,[,IOSTAT=ios][,ERR=s][,END=s])[list] 7.4.4

c is an internal file specifier.

f is a format specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

list is an I/O list.

Reads into elements in the list one or more internal records containing character strings, converting in accordance with the format specification.

RETURN [i] 4.7

Returns control to the calling program from the current subprogram. The optional argument is an integer value that indicates which alternate return is to be taken.

REWIND([UNIT=]u[,IOSTAT=ios][,ERR=s]) 9.4
REWIND u

u is a logical unit specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

Repositions logical unit u to the beginning of the currently opened file.

REWRITE([UNIT=]u,[FMT=]f[,IOSTAT=ios][,ERR=s])[list] 7.6

REWRITE([UNIT=]u[,IOSTAT=ios][,ERR=s])[list] 7.6

u is a logical unit specifier.

f is a format specifier.

ios is an input/output status specifier.

s is the label of an executable statement.

list is an I/O list.

Transfers data from internal storage to the current record in an indexed file.

Form	Effect	Manual Section
SAVE[a[,a]...]		5.6
	a — is the name of a variable, an array, or a named common block enclosed in slashes.	
	Retains the definition status of an entity after the execution of a RETURN or END statement in a subprogram.	
Statement Function		6.2.1
f([p[,p]...])=e		
	f — is a symbolic name.	
	p — is a symbolic name.	
	e — is an expression.	
	Creates a user-defined function having the variables p as dummy arguments. When referred to, the expression is evaluated using the actual arguments in the function call.	
STOP [disp]		4.9
	disp — is a decimal digit string containing 1 to 5 digits or a character constant.	
	Terminates program execution and prints the display, if one is specified.	
SUBROUTINE nam[([p[,p]...])]		6.2.3
	nam — is a symbolic name.	
	p — is a symbolic name or an alternate return specifier (*).	
	Begins a subroutine subprogram, indicating the program name and any dummy argument names (p).	
TYPE	See WRITE	7.8

Form	Effect	Manual Section

Type Declaration 5.2

typ v[/clist/][,v[/clist/]...

typ is one of the following data type specifiers:

BYTE
LOGICAL
LOGICAL*1
LOGICAL*2
LOGICAL*4
INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
REAL*16
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
DOUBLE COMPLEX
CHARACTER*len
CHARACTER*(*)

v is a variable name, array name, function or function entry name, or an array declarator. The name can optionally be followed by a data type length specifier (*n). For character entities, the length specifier can be *len or *(*).

clist is an initial value or values to be assigned to the immediately preceding variable or array element.

The symbolic names (v) are assigned the specified data type.

UNLOCK ([UNIT=]u[,IOSTAT=ios][,ERR=s]) 9.8

UNLOCK u 9.8

u is a logical unit specifier.

ios is an input/output specifier.

s is the label of an executable statement.

Removes the access protection from the file connected to Logical Unit u.

VIRTUAL a(d)[,a(d)]... 5.3

Equivalent to the DIMENSION statement.

Form	Effect	Manual Section
WRITE ([UNIT=]u,[FMT=]f[,IOSTAT=ios][,ERR=s])[list]		7.5.1.1
PRINT f[,list]		7.8
TYPE f[,list]		7.8

u — is a logical unit specifier.

f — is a format specifier.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Writes one or more logical records to unit u, containing the values of the elements in the list. The records are converted according to f.

WRITE([UNIT=]u,[FMT=]*[,IOSTAT=ios][,ERR=s])[list]		7.5.1.2
WRITE(u,*[,ERR=s])[list]		7.5.1.2
PRINT *[,list]		7.8
TYPE *[,list]		7.8

u — is a logical unit specifier.

* — denotes list-directed formatting.

ios — is an input/output specifier.

s — is a label of an executable statement.

list — is an I/O list.

Writes one or more logical records to unit u containing the values of the elements in the list. The records are converted according to the data type of the list element.

WRITE ([UNIT=]u[,IOSTAT=ios][,ERR=s])[list]	7.5.1.3

u — is a logical unit specifier.

s — is a label of an executable statement label.

ios — is an input/output specifier.

list — is an I/O list.

Writes one unformatted record to unit u containing the values of the elements in the list.

Form	Effect	Manual Section
WRITE ([UNIT=]u, REC=r, FMT=f[,IOSTAT=ios][,ERR=s])[list]		7.5.2.1
WRITE (u'r,f[,ERR=s])[list]		7.5.2.1
u	is a logical unit specifier.	
r	is a relative record number specifier.	
f	is a format specifier.	
ios	is an input/output specifier.	
s	is a label of an executable statement.	
list	is an I/O list.	
	Writes the values of the elements of the list to record r on unit u. The record is converted according to f.	
WRITE([UNIT=]u,REC=r[,IOSTAT=ios][,ERR=s])[list]		7.5.2.2
WRITE (u'r[,ERR=s]) [list]		7.5.2.2
u	is a logical unit specifier.	
r	is a relative record number specifier.	
ios	is an input/output specifier.	
s	is a label of an executable statement label.	
list	is an I/O list.	
	Writes record r to unit u containing the values of the elements in the list.	
WRITE ([UNIT=]c,FMT=f[,IOSTAT=ios][,ERR=s])[List]		7.5.4
c	is an internal file specifier.	
f	is a format specifier.	
s	is the label of an executable statement.	
ios	is an input/output specifier.	
list	is an I/O list.	
	Writes elements in the list to the internal file specified by the unit, converting to character strings in accordance with the format specification.	

C.3 LIBRARY FUNCTIONS

Table C-1 lists the VAX-11 FORTRAN generic functions and intrinsic functions (listed in the column headed "Specific Name"). Superscripts in the table refer to the notes that follow the table.

FORTRAN LANGUAGE SUMMARY

Table C-1 Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Square Root[1]	1	SQRT	SQRT	REAL*4	REAL*4
			DSQRT	REAL*8	REAL*8
			QSQRT	REAL*16	REAL*16
$a^{1/2}$			CSQRT	COMPLEX*8	COMPLEX*8
			CDSQRT	COMPLEX*16	COMPLEX*16
Natural Logarithm[2]	1	LOG	ALOG	REAL*4	REAL*4
			DLOG	REAL*8	REAL*8
			QLOG	REAL*16	REAL*16
$\log_e a$			CLOG	COMPLEX*8	COMPLEX*8
			CDLOG	COMPLEX*16	COMPLEX*16
Common Logarithm[2]	1	LOG10	ALOG10	REAL*4	REAL*4
			DLOG10	REAL*8	REAL*8
$\log_{10} a$			QLOG10	REAL*16	REAL*16
Exponential	1	EXP	EXP	REAL*4	REAL*4
e^a			DEXP	REAL*8	REAL*8
			QEXP	REAL*16	REAL*16
			CEXP	COMPLEX*8	COMPLEX*8
			CDEXP	COMPLEX*16	COMPLEX*16
Sine[3]	1	SIN	SIN	REAL*4	REAL*4
			DSIN	REAL*8	REAL*8
			QSIN	REAL*16	REAL*16
Sin a			CSIN	COMPLEX*8	COMPLEX*8
			CDSIN	COMPLEX*16	COMPLEX*16
Cosine[3]	1	COS	COS	REAL*4	REAL*4
			DCOS	REAL*8	REAL*8
			QCOS	REAL*16	REAL*16
Cos a			CCOS	COMPLEX*8	COMPLEX*8
			CDCOS	COMPLEX*16	COMPLEX*16
Tangent[3]	1	TAN	TAN	REAL*4	REAL*4
			DTAN	REAL*8	REAL*8
Tan a			QTAN	REAL*16	REAL*16
Arc Sine[4,5]	1	ASIN	ASIN	REAL*4	REAL*4
			DASIN	REAL*8	REAL*8
Arc Sin a			QASIN	REAL*16	REAL*16
Arc Cosine[4,5]	1	ACOS	ACOS	REAL*4	REAL*4
			DACOS	REAL*8	REAL*8
Arc Cos a			QACOS	REAL*16	REAL*16
Arc Tangent[5]	1	ATAN	ATAN	REAL*4	REAL*4
			DATAN	REAL*8	REAL*8
Arc Tan a			QATAN	REAL*16	REAL*16
Arc Tangent[5,6]	2	ATAN2	ATAN2	REAL*4	REAL*4
			DATAN2	REAL*8	REAL*8
Arc Tan a_1/a_2			QATAN2	REAL*16	REAL*16

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Hyperbolic Sine	1	SINH	SINH	REAL*4	REAL*4
			DSINH	REAL*8	REAL*8
Sinh a			QSINH	REAL*16	REAL*16
Hyperbolic Cosine	1	COSH	COSH	REAL*4	REAL*4
			DCOSH	REAL*8	REAL*8
Cosh a			QCOSH	REAL*16	REAL*16
Hyperbolic Tangent	1	TANH	TANH	REAL*4	REAL*4
			DTANH	REAL*8	REAL*8
Tanh a			QTANH	REAL*16	REAL*16
Absolute value[7]	1	ABS	IIABS	INTEGER*2	INTEGER*2
|a|			JIABS	INTEGER*4	INTEGER*4
			ABS	REAL*4	REAL*4
			DABS	REAL*8	REAL*8
			QABS	REAL*16	REAL*16
			CABS	COMPLEX*8	REAL*4
			CDABS	COMPLEX*16	REAL*8
		IABS	IIABS	INTEGER*2	INTEGER*2
			JIABS	INTEGER*4	INTEGER*4
Truncation[8,11]	1	INT	IINT	REAL*4	INTEGER*2
(a)			JINT	REAL*4	INTEGER*4
			IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
			IIQINT	REAL*16	INTEGER*2
			JIQINT	REAL*16	INTEGER*4
			—	COMPLEX*8	INTEGER*2
			—	COMPLEX*8	INTEGER*4
			—	COMPLEX*16	INTEGER*2
			—	COMPLEX*16	INTEGER*4
		IDINT	IIDINT	REAL*8	INTEGER*2
			JIDINT	REAL*8	INTEGER*4
		IQINT	IIQINT	REAL*16	INTEGER*2
			JIQINT	REAL*16	INTEGER*4
		AINT	AINT	REAL*4	REAL*4
			DINT	REAL*8	REAL*8
			QINT	REAL*16	REAL*16
Nearest Integer[8,11]	1	NINT	ININT	REAL*4	INTEGER*2
(a+.5*sign(a))			JNINT	REAL*4	INTEGER*4
			IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
			IIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		IDNINT	IIDNNT	REAL*8	INTEGER*2
			JIDNNT	REAL*8	INTEGER*4
		IQNINT	IIQNNT	REAL*16	INTEGER*2
			JIQNNT	REAL*16	INTEGER*4
		ANINT	ANINT	REAL*4	REAL*4
			DNINT	REAL*8	REAL*8
			QNINT	REAL*16	REAL*16

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to[9] REAL*4	1	REAL	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
			—	REAL*4	REAL*4
			SNGL	REAL*8	REAL*4
			SNGLQ	REAL*16	REAL*4
			—	COMPLEX*8	REAL*4
			—	COMPLEX*16	REAL*4
Conversion to[9] REAL*8	1	DBLE	—	INTEGER*2	REAL*8
			—	INTEGER*4	REAL*8
			DBLE	REAL*4	REAL*8
			—	REAL*8	REAL*8
			DBLEQ	REAL*16	REAL*8
			—	COMPLEX*8	REAL*8
			—	COMPLEX*16	REAL*8
Conversion to REAL*16	1	QEXT	—	INTEGER*2	REAL*16
			—	INTEGER*4	REAL*16
			QEXT	REAL*4	REAL*16
			QEXTD	REAL*8	REAL*16
			—	REAL*16	REAL*16
			—	COMPLEX*8	REAL*16
			—	COMPLEX*16	REAL*16
Fix[9,11]	1	IFIX	IIFIX	REAL*4	INTEGER*2
			JIFIX	REAL*4	INTEGER*4
(REAL*4-to-integer conversion)					
Float[9]	1	FLOAT	FLOATI	INTEGER*2	REAL*4
			FLOATJ	INTEGER*4	REAL*4
(integer-to-REAL*4 conversion)					
REAL*8 Float[9]	1	DFLOAT	DFLOTI	INTEGER*2	REAL*8
			DFLOTJ	INTEGER*4	REAL*8
(integer-to-REAL*8 conversion)					
REAL*16 Float	1	QFLOAT	—	INTEGER*2	REAL*16
				INTEGER*4	REAL*16
(Integer to REAL*16 conversion)					
Conversion to COMPLEX*8,	1,2[12]	CMPLX	—	INTEGER*2	COMPLEX*8
or	1,2		—	INTEGER*4	COMPLEX*8
COMPLEX*8 from Two	1,2		—	REAL*4	COMPLEX*8
Arguments	1,2		—	REAL*8	COMPLEX*8
	1,2		—	REAL*16	COMPLEX*8
	1		—	COMPLEX*8	COMPLEX*8
	1		—	COMPLEX*16	COMPLEX*8

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Conversion to COMPLEX*16, or COMPLEX*16 from Two Arguments	1,2[12]	DCMPLX	—	INTEGER*2	COMPLEX*16
	1,2		—	INTEGER*4	COMPLEX*16
	1,2		—	REAL*4	COMPLEX*16
	1,2		—	REAL*8	COMPLEX*16
	1,2		—	REAL*16	COMPLEX*16
	1		—	COMPLEX*8	COMPLEX*16
	1		—	COMPLEX*16	COMPLEX*16
Real Part of Complex	1	—	REAL	COMPLEX*8	REAL*4
			DREAL	COMPLEX*16	REAL*8
Imaginary Part of Complex	1	—	AIMAG	COMPLEX*8	REAL*4
			DIMAG	COMPLEX*16	REAL*8
Complex From Two Arguments	(See Conversion to COMPLEX*8 and Conversion to COMPLEX*16)				
Complex Conjugate (if a=(X,Y) CONJG (a)=(X,-Y)	1	CONJG	CONJG	COMPLEX*8	COMPLEX*8
			DCONJG	COMPLEX*16	COMPLEX*16
REAL*8 product of REAL*4's a_1*a_2	2	—	DPROD	REAL*4	REAL*8
Maximum[11] $max(a_1,a_2,...a_n)$ (returns the maximum value from among the argument list; there must be at least two arguments)	n	MAX	IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
			AMAX1	REAL*4	REAL*4
			DMAX1	REAL*8	REAL*8
			QMAX1	REAL*16	REAL*16
		MAX0	IMAX0	INTEGER*2	INTEGER*2
			JMAX0	INTEGER*4	INTEGER*4
		MAX1	IMAX1	REAL*4	INTEGER*2
			JMAX1	REAL*4	INTEGER*4
		AMAX0	AIMAX0	INTEGER*2	REAL*4
			AJMAX0	INTEGER*4	REAL*4
Minimum[11] $min(a_1,a_2,...a_n)$ (returns the minimum value among the argument list; there must be at least two arguments)	n	MIN	IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
			AMIN1	REAL*4	REAL*4
			DMIN1	REAL*8	REAL*8
			QMIN1	REAL*16	REAL*16
		MIN0	IMIN0	INTEGER*2	INTEGER*2
			JMIN0	INTEGER*4	INTEGER*4
		MIN1	IMIN1	REAL*4	INTEGER*2
			JMIN1	REAL*4	INTEGER*4
		AMIN0	AIMIN0	INTEGER*2	REAL*4
			AJMIN0	INTEGER*4	REAL*4

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Positive Difference $a_1-(\min(a_1,a_2))$ (returns the first argument minus the minimum of the two arguments)	2	DIM	IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
			DIM	REAL*4	REAL*4
			DDIM	REAL*8	REAL*8
			QDIM	REAL*16	REAL*16
		IDIM	IIDIM	INTEGER*2	INTEGER*2
			JIDIM	INTEGER*4	INTEGER*4
Remainder $a_1-a_2*[a_1/a_2]($ returns the remainder when the first argument is divided by the second)	2	MOD	IMOD	INTEGER*2	INTEGER*2
			JMOD	INTEGER*4	INTEGER*4
			AMOD	REAL*4	REAL*4
			DMOD	REAL*8	REAL*8
			QMOD	REAL*16	REAL*16
Transfer of Sign $\|a_1\|$*Sign a_2	2	SIGN	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
			SIGN	REAL*4	REAL*4
			DSIGN	REAL*8	REAL*8
			QSIGN	REAL*16	REAL*16
		ISIGN	IISIGN	INTEGER*2	INTEGER*2
			JISIGN	INTEGER*4	INTEGER*4
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND	INTEGER*2	INTEGER*2
			JIAND	INTEGER*4	INTEGER*4
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR	INTEGER*2	INTEGER*2
			JIOR	INTEGER*4	INTEGER*4
Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR	INTEGER*2	INTEGER*2
			JIEOR	INTEGER*4	INTEGER*4
Bitwise Complement (complements each bit)	1	NOT	INOT	INTEGER*2	INTEGER*2
			JNOT	INTEGER*4	INTEGER*4
Bitwise Shift (a_1 logically shifted left a_2 bits)	2	ISHFT	IISHFT	INTEGER*2	INTEGER*2
			JISHFT	INTEGER*4	INTEGER*4
Length[11] (returns length of the character expression)	1	—	LEN	CHARACTER	INTEGER*4
Index (C_1, C_2)[11] (returns the position of the substring c_2 in the character expression c_1)	2	—	INDEX	CHARACTER	INTEGER*4
Character[11] (returns a character that has the ASCII value specified by the argument)	1	—	CHAR	LOGICAL*1 INTEGER*2 INTEGER*4	CHARACTER

Table C-1 (Cont.) Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
ASCII Value[10] (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	—	ICHAR	CHARACTER	INTEGER*4
Character relationals (ASCII collating sequence)	2	—	LLT	CHARACTER	LOGICAL*4
	2	—	LLE	CHARACTER	LOGICAL*4
	2	—	LGT	CHARACTER	LOGICAL*4
	2	—	LGE	CHARACTER	LOGICAL*4

Notes for Table C-1

[1] The argument of SQRT, DSQRT, and QSQRT must be greater than or equal to zero. The result of CSQRT or CDSQRT is the principal value with the real part greater than or equal to zero. When the real part is zero, the result is the principal value with the imaginary part greater than or equal to zero.

[2] The argument of ALOG, DLOG, QSQRT, ALOG10, DLOG10, and QLOG10 must be greater than zero. The argument of CLOG or CDLOG must not be (0.,0.).

[3] The argument of SIN, DSIN, QSIN, COS, DCOS, QCOS, TAN, DTAN, and QTAN must be in radians. The argument is treated modulo 2*pi.

[4] The absolute value of the argument of ASIN, DASIN, QASIN, ACOS, DACOS and QACOS must be less than or equal to 1.

[5] The result of ASIN, DASIN, QASIN, ACOS, DACOS, QACOS, ATAN, DATAN, QATAN, ATAN2, DATAN2, and QATAN2 is in radians.

[6] The result of ATAN2, DATAN2, and QATAN2 is zero or positive when a_2 is less than or equal to zero. The result is undefined if both arguments are zero.

[7] The absolute value of a complex number, (X,Y), is the real value:

$(X^2+Y^2)^{1/2}$

[8] [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [-5.7] equals -5.

[9] Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function REAL with a real argument, the function DBLE with a double precision argument, the function INT with an integer argument, and the function QEXT with a REAL*16 argument return the value of the argument without conversion.

[10] See Chapter 6 for additional information on character functions.

[11] The functions INT, IDINT, IQINT, NINT, IDNINT, IQNINT, IFIX, MAXI, and MINI return INTEGER*4 values if the /I4 command qualifier is in effect, INTEGER*2 values if the /NOI4 qualifier is in effect.

[12] When CMPLX and DCMPLX have only one argument, this argument is converted into the real part of a complex value, and zero is assigned to the imaginary part; when there are two arguments (not complex), a complex value is produced by conversion of the first argument into the real part of the value, the second argument into the imaginary part.

C.4 SYSTEM SUBROUTINE SUMMARY

The VAX-11 FORTRAN system provides subroutines you call in the same manner as a user-written subroutine. These subroutines are described in this section.

The subroutines supplied are:

DATE	Returns a 9-byte string containing the ASCII representation of the current date.
IDATE	Returns three integer values representing the current month, day, and year.
ERRSNS	Returns information about the most recently detected error condition.
EXIT	Terminates the execution of a program and returns control to the operating system.
SECNDS	Provides system time of day, or elapsed time, as a floating-point value in seconds.
TIME	Returns an 8-byte string containing the ASCII representation of the current time in hours, minutes, and seconds.
RAN	Returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1.

References to integer arguments in the following subroutine descriptions refer to arguments of either INTEGER*4 data type or INTEGER*2 data type. However, the arguments must be either all INTEGER*4 or all INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines if their value is within the INTEGER*2 range.

C.4.1 DATE

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form

```
CALL DATE(buf)
```

where:

buf
is a 9-byte variable, array, array element, or character substring. The date is returned as a 9-byte ASCII character string of the form

```
dd-mmm-yy
```

where:

dd
is the 2-digit date.

mmm
is the 3-letter month specification.

yy
is the last two digits of the year.

C.4.2 IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. The call to IDATE has the form

```
CALL IDATE(i,j,k)
```

If the current date were October 9, 1984, the values of the integer variables upon return would be:

```
i = 10
j = 9
k = 84
```

C.4.3 ERRSNS

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form

```
CALL ERRSNS(fnum,rmssts,rmsstv,iunit,condval)
```

where:

fnum
is an integer variable or array element into which is stored the most recent FORTRAN error number. VAX-11 FORTRAN error numbers are listed in the VAX-11 FORTRAN User's Guide.

A zero is returned if no error has occurred since the last call to ERRSNS, or if no error has occurred since the start of execution.

rmssts
is an integer variable or array element into which is stored the RMS completion status code (STS), if the last error was an RMS I/O error.

rmsstv
is an integer variable or array element into which is stored the RMS status value (STV) if the last error was an RMS I/O error. This status value provides additional status information.

iunit
is an integer variable or array element into which is stored the logical unit number, if the last error was an I/O error.

condval
is an integer variable or array element into which is stored the actual VAX-11 condition value.

Any of the arguments may be null. If the arguments are of INTEGER*2 type, only the low-order 16 bits of information are returned. The saved error information is set to zero after each call to ERRSNS.

C.4.4 EXIT

The EXIT subroutine causes program termination, closes all files, and returns control to the operating system. A call to EXIT has the form

```
CALL EXIT [(exit-status)]
```

where:

(exit-status)

is an optional integer argument you can use to specify the image exit-status value.

C.4.5 SECNDS

The SECNDS function subprogram returns the system time in seconds as a single-precision, floating-point value less the value of its single-precision, floating-point argument. The call to SECNDS has the form

```
y = SECNDS(x)
```

where:

y

is set equal to the time in seconds since midnight, minus the user-supplied value of x.

The SECNDS function can be used to perform elapsed-time computations. For example:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)

C
C     CODE TO BE TIMED
C
      DELTA = SECNDS(T1)
```

where:

DELTA

will give the elapsed time.

The value of SECNDS is accurate to 0.01 seconds, which is the resolution of the system clock.

NOTES

1. The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.

2. The 24 bits of precision provides accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day. More precise timing information can be obtained using Run Time Library procedures:

```
LIB$INIT_TIMER
LIB$SHOW_TIMER
LIB$STAT_TIMER
```

C.4.6 TIME

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form

```
CALL TIME(buf)
```

where buf is an 8-byte variable, array, array element, or character substring.

The TIME call returns the time as an 8-byte ASCII character string of the form

```
hh:mm:ss
```

where:

hh
is the 2-digit hour indication.

mm
is the 2-digit minute indication.

ss
is the 2-digit second indication.

For example:

```
10:45:23
```

A 24-hour clock is used.

C.4.7 RAN

The RAN function is a general random number generator of the multiplicative congruential type. The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. The call to RAN has the form:

```
y=RAN(i)
```

where:

y

is set equal to the value associated, by the function, with the argument i. The argument i must be an INTEGER*4 variable or INTEGER*4 array element.

The argument should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers; the seed is updated automatically. RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 (MOD 2**32)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z