August 1978

This document describes the FORTRAN language elements supported by VAX-11 FORTRAN IV-PLUS. 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 IV-PLUS
Language Reference Manual

Order No. AA-D034A-TE

SUPERSESSION/UPDATE INFORMATION: This is a new document for this release.

OPERATING SYSTEM AND VERSION: VAX/VMS V01

SOFTWARE VERSION: VAX-11 FORTRAN IV-PLUS 01

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 1978

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	

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the elements of VAX-11 FORTRAN IV-PLUS. The manual is designed for reference, rather than as a tutorial document.

The user interface to VAX-11 FORTRAN IV-PLUS is described in the companion manual to this document, the VAX-11 FORTRAN IV-PLUS User's Guide.

INTENDED AUDIENCE

This manual is intended for use as a reference document. Therefore, readers should have a basic understanding of the FORTRAN language in order to derive maximum benefit from the manual. Some knowledge of the VAX/VMS operating system is helpful, but not necessary. For information concerning VAX/VMS refer to the documents listed below under "ASSOCIATED DOCUMENTS."

STRUCTURE OF THIS DOCUMENT

This manual contains nine chapters and two appendixes.

- Chapter 1 consists of general information concerning FORTRAN, and introduces basic facts needed prior to writing FORTRAN programs.
- Chapter 2 describes the components of FORTRAN statements, such as symbols, constants, variables, etc.
- Chapter 3 describes assignment statements, which are used to define values used in the program.
- Chapter 4 deals with control statements, used to transfer control from one point in the program to another.
- Chapter 5 describes specification statements, which are used to define characteristics of symbols used in the program, such as data type, array dimensions, etc.
- Chapter 6 discusses subprograms; both user-written and those supplied with VAX-11 FORTRAN IV-PLUS.
- Chapter 7 covers the topic of FORTRAN input/output.
- Chapter 8 describes the FORMAT statements used in conjunction with formatted I/O statements.

- Chapter 9 contains information on auxiliary I/O statements, such as OPEN, CLOSE, and DEFINE FILE.
- Appendix A summarizes the character sets supported by VAX-11 FORTRAN IV-PLUS.
- Appendix B summarizes the language elements of VAX-11 FORTRAN IV-PLUS.

ASSOCIATED DOCUMENTS

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

- VAX/VMS Primer
- VAX-11 FORTRAN IV-PLUS 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:

- Uppercase words and letters used in examples indicate that you should type the word or letter as shown
- Lowercase 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

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

Tab character TAB

Space character Δ

or

SP (in terminal dialog examples)

CHAPTER 1

INTRODUCTION TO VAX-11 FORTRAN IV-PLUS

1.1 LANGUAGE OVERVIEW

VAX-11 FORTRAN IV-PLUS[1] is based on American National Standard (ANS) FORTRAN X3.9-1966, and includes the following enhancements to ANS FORTRAN:

- The CHARACTER data type provides a means of manipulating character data. Character constants, variables, arrays, functions, and expressions using the concatenation operator and substring references are provided.

- Block IF statements provide a way to conditionally execute a block of statements. The block IF statements are IF THEN, ELSE IF THEN, ELSE, and END IF.

- Any arithmetic expression can be used as an array subscript. If the expression is not of type integer 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
CHARACTER
```

- The IMPLICIT statement redefines the implied data type of symbolic names.

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

```
ACCEPT
TYPE                    Device-oriented I/O
PRINT

READ  (u'r)
WRITE (u'r)             Unformatted direct-access I/O
FIND  (u'r)

READ  (u'r,fmt)
WRITE (u'r,fmt)         Formatted direct-access I/O
```

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

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

- The specifications END=s and/or ERR=s can be included in any READ or WRITE statement to transfer control to the specified statement when an end-of-file or error condition occurs.
- List-directed I/O can be used to perform formatted I/O without a format specification.
- Constants and expressions are permitted in the I/O lists of WRITE, TYPE, and PRINT statements.
- PARAMETER statements can be used to assign symbolic names to constant values.
- Generic-function selection by argument data type is provided for many FORTRAN-supplied functions.
- A DO statement control variable can be a real or double precision variable. Any arithmetic expression can be used as the initial value, increment, or limit parameter in the DO statement, or as the control parameter in the computed GO TO statement.
- The DO statement increment parameter can have a negative value.
- For readability, you can optionally use commas in DO statements. For example:

```
DO 5, I=1,10
```

- Lower bounds for array dimensions can be specified in all array declarators.
- ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.
- A PROGRAM statement can be used in a main program.
- 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 as any FORTRAN statement. Begin comments 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 you specify the associated compiler command qualifier; otherwise, they are treated as comments.
- The statement label list in an assigned GO TO statement is optional.

- Octal and hexadecimal constants can be used in place of any numeric constants.
- Symbolic names can be up to 15 characters long and consist of letters, digits, dollar signs ($), and underline characters (_).

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

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. 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.

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 in the first column of a source line identifies that line as a comment. 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 uppercase and lowercase letters (A through Z, a through z)
2. The numerals 0 through 9
3. The special characters listed below

Character	Name
Δ or (TAB)	Space or tab
=	Equal sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Period
'	Apostrophe
"	Quotation mark
$	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 A 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 uppercase and lowercase letters.

1.3 FORMATTING A FORTRAN LINE

Each FORTRAN line has four fields, as follows:

- 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 using a text editor. You can use tab-character formatting only when you are entering lines at a terminal using 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.

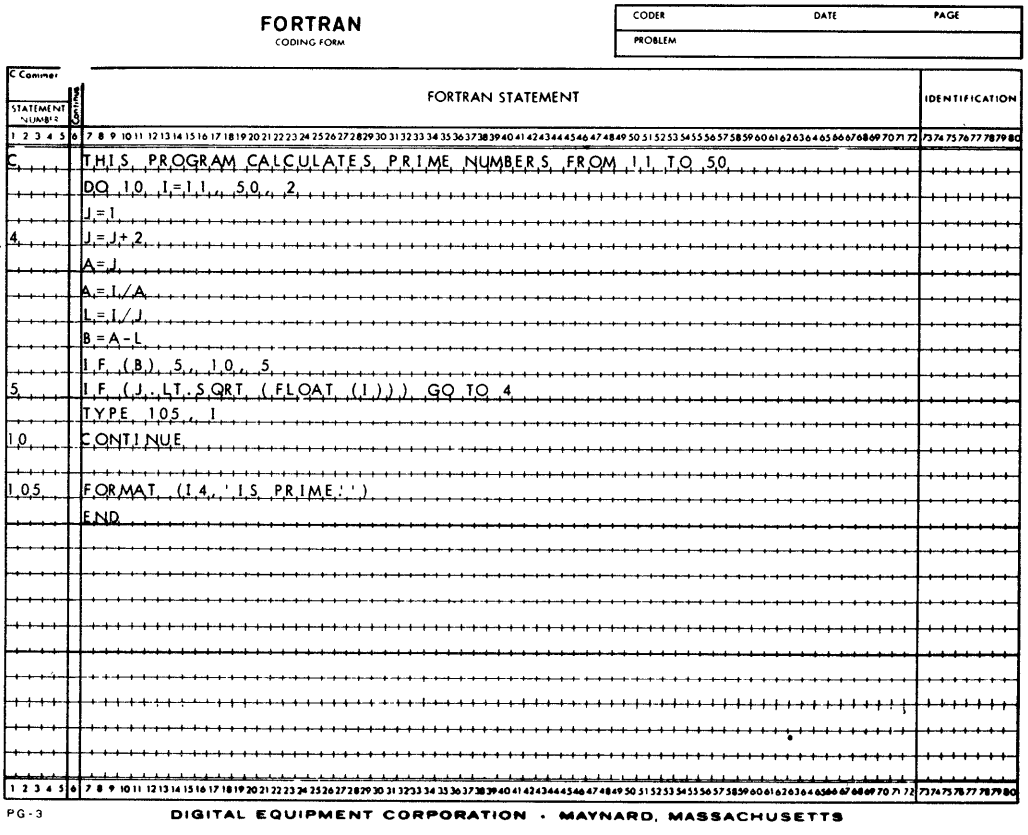


Figure 1-1 FORTRAN Coding Form

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. 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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C						F	I	R	S	T		V	A	L	U	E			
1	0					I		=		J		+		5	*	K		+	
					1		L	*	M										
						I	V	A	L		=		I	+	2				

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 5 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

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 illegal.

Any statement that another statement refers to must have a label. 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 in column 1 to indicate that the line is a comment. The compiler prints that line in the source program listing, then ignores the line.

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 as well as 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. A statement can be divided into distinct lines at any point. The compiler considers the characters after the continuation character as the characters following the last character of the previous line, as if no break occurred at that point. If a continuation indicator is zero, then the compiler considers the line an initial line of a FORTRAN statement.

Comment lines cannot be continued. Comment lines 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. Horizontal lines indicate statement types that cannot be interspersed. For example, type declaration statements cannot be interspersed with executable statements.

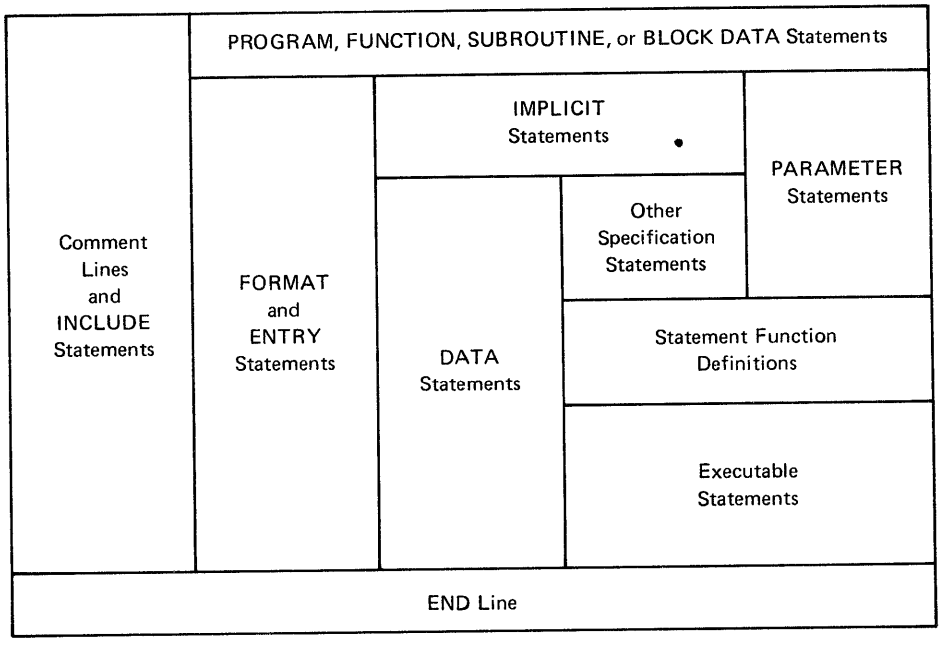


Figure 1-3 Required Order of Statements and Lines

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]'
```

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 IV-PLUS 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 that a comment line can appear, 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))
           .
           .
           .

      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 data types:

- Logical
- Integer
- Real
- Double precision
- Complex
- Character

The following 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 are used to 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 15 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
Arithmetic statement functions	yes
Processor-defined 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 -- a decimal number; that is, a whole number, a decimal fraction, or a combination of the two
- Double precision -- similar to real, but with more than twice the degree of accuracy in its representation
- Complex -- a pair of real values that represent a complex number; the first value represents the real part of that number, the second represents the imaginary part
- Logical -- the logical value, true or false
- Character -- a sequence of characters

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.

ANS FORTRAN specifies that a "numeric storage unit" is the amount of storage needed to represent a real, integer, or logical value. Double precision and complex values occupy two numeric storage units. In VAX-11 FORTRAN IV-PLUS, a numeric storage unit corresponds to 4 bytes of memory.

ANS FORTRAN specifies that a "character storage unit" is the amount of storage needed to represent one character value. In VAX-11 FORTRAN IV-PLUS, a character storage unit corresponds to 1 byte of memory.

VAX-11 FORTRAN IV-PLUS 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[a]
LOGICAL	2 or 4[b]
LOGICAL*1	1[a]
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 or 4[b]
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
COMPLEX	8
COMPLEX*8	8
CHARACTER*len	len[c]
CHARACTER*(*)	

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

[b] Either 2 or 4 bytes are allocated depending on the compiler command qualifier specified. The default allocation is 4 bytes.

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

The following sections contain additional descriptions of these data types and their representations.

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 the data type of 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

s
An optional sign.

nn
A string of numeric characters.

Leading zeros, if any, are ignored.

A minus sign must appear before a negative integer constant. 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 absolute value of an integer constant cannot be greater than 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.

The octal form of an integer constant is:

"nn

nn
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)

2.3.2 Real Constants

A real 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

A basic real constant is a string of decimal digits, in any of the following forms:

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

s
An optional sign

nn
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 7 digits are significant. Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting the leftmost 7 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
```

s
An optional sign.

nn
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 constant occupies 4 bytes of VAX-11 storage and is interpreted as a real number with a degree of precision of, typically, 7 decimal digits.

A minus sign must appear between the letter E and a negative exponent. A plus sign is optional for a positive exponent.

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

If the letter E appears in a real constant, an integer constant exponent field must follow. The exponent field cannot be omitted, but it can be zero.

The magnitude of a non-zero real constant cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Examples of valid and invalid real constants are:

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 Double Precision Constants

A double precision constant is a basic real constant or an integer constant followed by a decimal exponent of the form:

Dsnn

s
An optional sign.

nn
An integer constant.

The number of digits that precede the exponent is not limited, but typically only the leftmost 16 digits are significant.

A double precision constant occupies 8 bytes of VAX-11 storage and is interpreted as a real number with a degree of precision that is typically 16 significant digits.

A minus sign must appear before a negative double precision constant. A plus sign is optional before a positive constant. Similarly, a minus sign must appear between the letter D and a negative exponent; a plus sign is optional for a positive exponent.

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

The magnitude of a non-zero double precision constant cannot be less than approximately 0.29D-38 or greater than approximately 1.7D38.

Examples of valid and invalid double precision constants are:

Valid	Invalid	
1234567890D+5	1234567890D45	(too large)
+2.7182818284846182D00	1234567890.0D-89	(too small)
-72.5D-15	+2.7182812846182	(no Dsnn present; this is a valid single precision constant)
1D0		

2.3.4 Complex Constants

A complex constant is a pair of real constants separated by a comma and enclosed in parentheses. The first real constant represents the real part of that number and the second real constant represents the imaginary part.

A complex constant has the form:

(rc,rc)

rc
A real constant.

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

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

Examples of valid and invalid complex constants are:

Valid	Invalid
(1.70391,-1.70391)	(1,2) (integers are not allowed)
(+12739E3,0.)	(1.23,) (second real constant is missing)
	(1.0,1.000) (double precision constants are not allowed)

2.3.5 Octal and Hexadecimal Constants

Octal and hexadecimal constants are alternative ways to represent numeric constants. They can be used 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:

$'c_1c_2c_3...c_n'O$

c
A digit in the range 0 to 7.

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

$'c_1c_2c_3...c_n'X$

c
A 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. Octal constants must be in the range '0'O to '37777777777'O, and hexadecimal constants must be in the range '0'X to 'FFFFFFFF'X.

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.

- 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:

```
                                    Data Type    Length of
      Statement                     of Constant  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 is required, generally integer, that type is assumed for the constant. For example:

```
                                    Data Type    Length of
      Statement                     of Constant  Constant

Y(IX)=Y('15'O)+3.                   INTEGER*4       4
```

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

```
                                    Data Type    Length of
      Statement                     of Constant  Constant

CALL APAC('34BC2'X)                 none            4
```

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

```
                                    Data Type    Length of
      Statement                     of Constant  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 4 bytes of data. When the data type implies that the length of the constant is more than 4 bytes, the leftmost digits have a value of 0. When the data type implies that the length of the constant is less than 4 bytes, the constant is truncated on the left. Table 2-2 (in Section 2.2) lists the number of bytes that each data type requires.

2.3.6 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.

2.3.7 Character Constants

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

A character constant has the form:

$'c_1c_2c_3...c_n'$

c
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 255.

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 1 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 2.3.8.

2.3.8 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:

$$nHc_1c_2c_3...c_n$$

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

c
: A printable character.

The maximum number of characters is 255.

Hollerith constants are stored as byte strings, 1 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; they cannot be used where a character value is expected.

Examples of valid and invalid Hollerith constants are:

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

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:

```
INTEGER*2 ICOUNT
REAL*8 DOUBLE
```

Statement	Data Type of Constant	Length of Constant
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.

Table 2-2 (in Section 2.2) lists the number of characters required for each data type. Each character occupies 1 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 the 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 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 cation.

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 NUMBER 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 the array name (see Section 2.5.4). An array element or an entire array can be defined before program execution by a DATA statement. An array element can be defined during program execution by an assignment or input statement; and an entire array can be defined during program execution by 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] ...)
```

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

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

```
[dl:]du
```

dl
The lower bound of the dimension.

du
The upper bound of the 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 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.

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
- Each operator is a +, -, *, /, or ** operator

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 bounds as well as the array name as subprogram arguments. 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]...)
```

s
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 is not of type integer, it is converted to an integer value by truncation of any fractional part before use.

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.

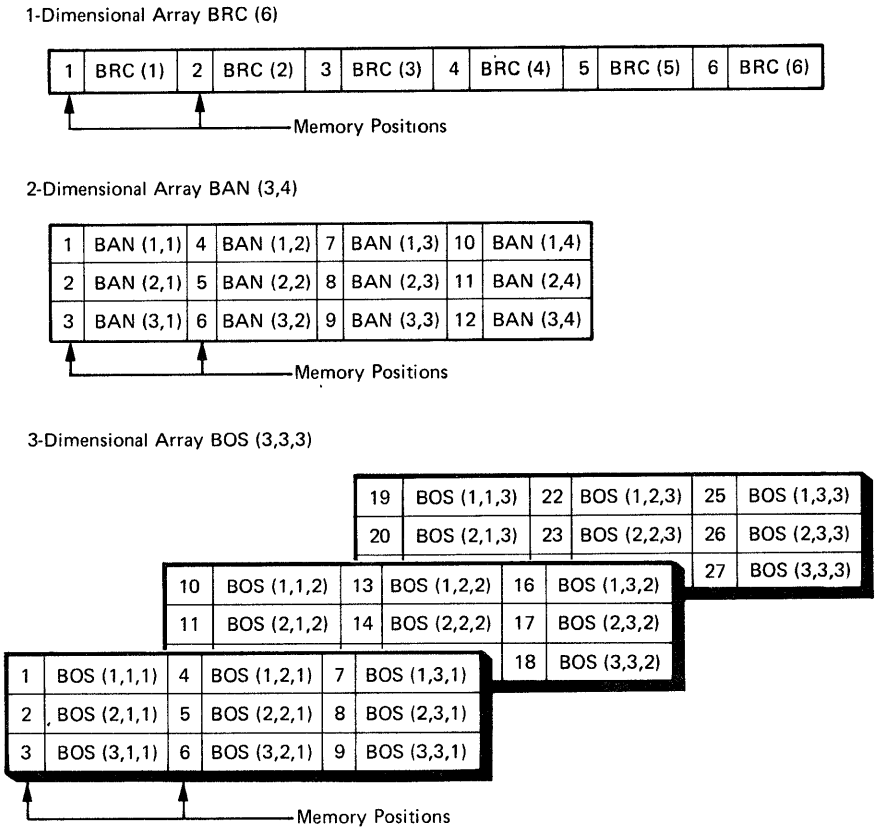


Figure 2-1 Array Storage

2.5.4 Data Type of an Array

The data type of an array is specified 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.

2.5.5 Array References Without Subscripts

In the following types of statement, 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
- 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 its name, as subprogram arguments. 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])
```

v
A character variable name.

a
A character array name.

s
A subscript expression.

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

e2
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 truncation of any fractional part before use.

The value of the numeric expressions, e1 and e2, must be such that:

```
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 constant
- A numeric variable
- A numeric array element
- An arithmetic expression enclosed in parentheses
- An arithmetic function reference

The term "numeric," as used above, can also be interpreted to include logical data, since 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.

Any arithmetic operator can be used with any valid arithmetic element, except as noted in Table 2-3.

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

Table 2-3 shows the allowed combinations and result data types of base and exponent data types for the exponentiation operator.

Table 2-3
Allowed Combinations for Exponentiation

Base	Exponent			
	Integer	Real	Double	Complex
Integer	Integer	No	No	No
Real	Real	Real	Double	No
Double	Double	Double	Double	No
Complex	Complex	No	No	No

Note: A negative element can be exponentiated only by an integer element; and an element with a zero value cannot be exponentiated by another zero-value element.

In any valid exponentiation, the result has the same data type as the base element, except in the case of a real base and a double precision exponent. The result in this case is double precision.

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, 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
Double precision	5
Complex	6 (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.

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 as follows:

- 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:

  ```
  1/3 + 1/3 + 1/3
  ```

  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 0. 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.

- Double precision operations -- Any real or integer element in a double precision operation is converted to double precision data type by making the existing element the most significant portion of a double precision datum. The least significant portion is 0. The expression is then evaluated in double precision arithmetic.

- Converting a real element to a double precision element does not increase its accuracy. For example, the real number

  0.3333333

  is converted to

  0.3333333000000000D0

  not to

  0.3333333333333333D0

- Complex operations -- In an operation that contains any complex element, integer elements are converted to real data type, as previously described. Double precision elements are converted to real data type by rounding the least significant portion. The real element thus obtained is designated as the real part of a complex number; the imaginary part is 0. The expression is then evaluated using complex arithmetic and the resulting value is of complex data type.

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 enclosed in parentheses
- A character function reference

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

A character expression is a sequence of one or more character elements separated by 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, 'ABCΔ'//'DΔE'//'FΔ' has a value of 'ABCΔDΔEFΔ'.

The order in which the character elements of a character expression are evaluated is determined by the compiler even if parentheses are present.

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 or not 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'. That relationship does exist, so 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.  'ABCΔΔΔ'

'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. Arithmetic and character operators have a higher precedence than relational operators.

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

Two numeric expressions of different data types can be compared by a relational expression. 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.
.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, cannot 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.	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 assignment statements are:

- Arithmetic assignment statement
- Logical assignment statement
- Character assignment statement
- ASSIGN statement

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
```

v
A numeric variable or array element.

e
An 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.

If the variable or array element on the left of the equal sign has the same data type as 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, Logical, or Octal Constant	Real	Double Precision	Complex
Integer or Logical	Assign E 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
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 real part of E to V; imaginary part of E is not used
Double Precision	Append fraction (.0) to E and assign to MS* portion of V; LS* portion of V is 0	Assign E to MS* portion of V; LS* portion of V is 0	Assign E to V	Assign real part of E to MS* portion of V; LS* portion of V is zero, 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 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'           (expression on 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
```

v
A logical variable or array element.

e
A logical expression.

The variable or array element on the left of the equal sign must be of logical data type. Its value can be undefined.

Values, either numeric or logical, 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
```

v
A character variable, array element, or substring.

e
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.

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.

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 that are 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 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 left must be a character variable,
                     array element, or substring reference)

CHARS = 25          (expression on 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 to specify a transfer destination in a subsequent assigned GO TO statement (see Section 4.1.3).

The ASSIGN statement has the form:

```
ASSIGN s TO v
```

s
The label of an executable statement in the same program unit as the ASSIGN statement.

v
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 assigned GO TO statement(s) in which the assigned variable is to be used. The ASSIGN statement and the assigned GO TO statement(s) 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, since a statement label cannot be altered.

```
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 interrupt normal program flow 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 statement -- specifies iterative processing
- 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 use.

GO TO

4.1 GO TO STATEMENTS

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

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
```

s
The label of an executable statement in the same program unit as 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 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.

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

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

e
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. The relationship between the variable and a specific statement label must be established by an ASSIGN statement. 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)]
```

v
An integer variable.

slist
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. Statements to which control is transferred must be executable statements in the same program unit.

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 statement
- Logical IF statement
- Block IF statements (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
```

e
An arithmetic expression.

s1,s2,s3
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. You can use one or two labels to refer to the statement immediately after the IF statement.

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
```

e
A logical expression.

st
A complete FORTRAN statement. The statement can be any executable statement except a DO statement, a block IF statement, an END 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:

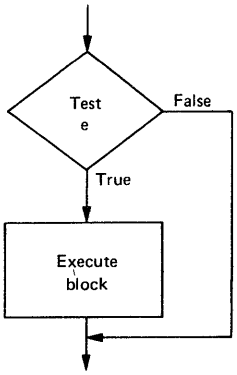
```
IF (e) THEN
   block

ELSE IF (e) THEN
   block
    .
    .
    .

ELSE
   block

END IF
```

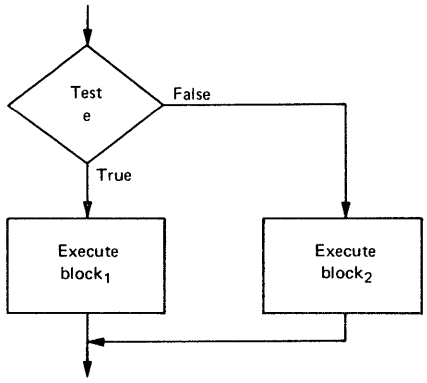
e
A logical expression.

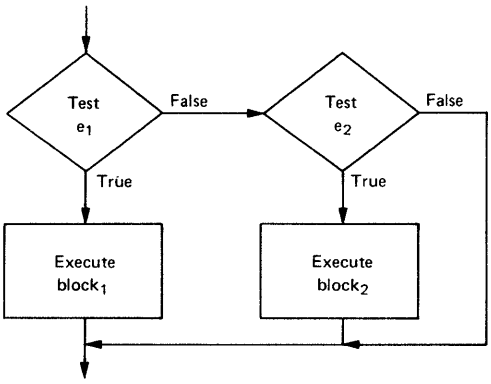
block
A sequence of zero or more complete FORTRAN statements. This sequence is called a statement block.

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

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
	

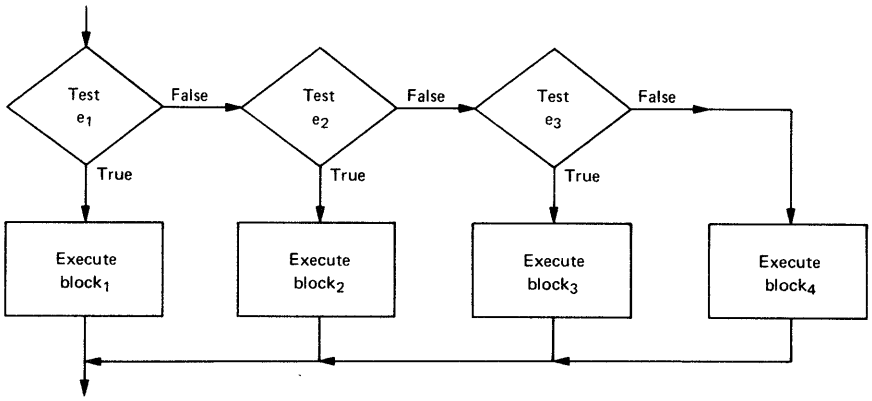
IF (e_1) THEN
$block_1$
ELSE IF (e_2) THEN
$block_2$
ELSE IF (e_3) THEN
$block_3$
ELSE
$block_4$
END IF
	

Figure 4-1 Examples of Block IF Constructs

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. No block IF statement can follow the ELSE statement except the END IF statement. The ELSE statement is optional.

The END IF statement terminates the block IF construct. The END IF statement is required.

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

The ELSE IF THEN and ELSE statements cannot have statement labels. The END IF statement can have a statement label, but control can only be transferred to its label 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. 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 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
  block1
	IF (NAME .LT. 'N') THEN
	  IFRONT = IFRONT + 1
	  FRLET(IFRONT)=NAME(1:2)

ELSE
  block2
	ELSE
	  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
  block1
	IF (A .GT. B) THEN
	  D = B
	  F = A - B

ELSE IF (e2) THEN
  block2
	ELSE IF (A .GT. C) THEN
	  D = C
	  F = A - C

ELSE IF (e3) THEN
  block3
	ELSE IF (A .GT. Z) THEN
	  D = Z
	  F = A - Z

ELSE
  block4
	ELSE
	  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	Example
IF (e) THEN	IF (A .LT. 100) THEN

block1 { IF (e) THEN
  blocka
ELSE
  blockb
END IF
	INRAN=INRAN + 1
	IF (ABS (A-AVG) .LE. 5.) THEN
	  INAVG = INAVG + 1
	ELSE
	  OUTAVG = OUTAVG + 1
	END IF

ELSE
  block2
	ELSE
	  OUTRAN = OUTRAN + 1

END IF	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 DO statement specifies 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]
```

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

v
An integer, real, or double precision variable.

e1,e2,e3
Arithmetic expressions.

The variable v is the control variable; and 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 terminal statement of a DO loop is identified by the label that appears in the DO statement. The terminal statement must not be one of the following statements:

- GO TO statement
- Arithmetic IF statement
- Any block IF statement, except ENDIF
- 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.

The DO statement first evaluates the expressions e1, e2, and e3 to determine values for the initial, terminal, and increment parameters. The initial, terminal, and increment parameters are converted, before use, to the data type of the control variable, if necessary. The value of the initial parameter is assigned to the control variable. The executable statements in the range of the DO loop are then executed repeatedly.

If the increment parameter (e3) is positive, the terminal parameter (e2) must be greater than or equal to the initial parameter (e1). Conversely, if e3 is negative, e1 must be less than or equal to e2. The increment parameter (e3) cannot be zero.

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

$$\left[\frac{e2 - e1}{e3}\right] + 1$$

where [X] (the value of the expression) 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 loop is executed once.

4.3.1 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.

Note that if the data type of the control variable is real or double precision, 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, if other DO loops share its terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (Section 4.3.2). 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, terminating, 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 4.3.2 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 4.3.3 and 4.3.4.

Examples of DO iteration control follow.

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

This statement specifies 25 iterations; K=49 during the final iteration.

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

This statement specifies 27 iterations; J=-2 during the final iteration.

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

This statement specifies 5 iterations; IVAR=5 during the final iteration.

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

This is an invalid statement; the statement label is missing.

```
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
```

4.3.2 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 terminal statement.

Figure 4-2 illustrates nested loops.

Correctly Nested DO Loops	Incorrectly Nested DO Loops
DO 45 K=1,10 . . DO 35 L=2,50,2 . . 35 CONTINUE . . DO 45 M=1,20 . . 45 CONTINUE	DO 15 K=1,10 . . DO 25 L=1,20 . . 15 CONTINUE . . DO 30 M=1,15 . . . 25 CONTINUE . . 30 CONTINUE

Figure 4-2 Nested DO Loops

4.3.3 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.

4.3.4 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 illustrates valid and invalid extended range control transfers.

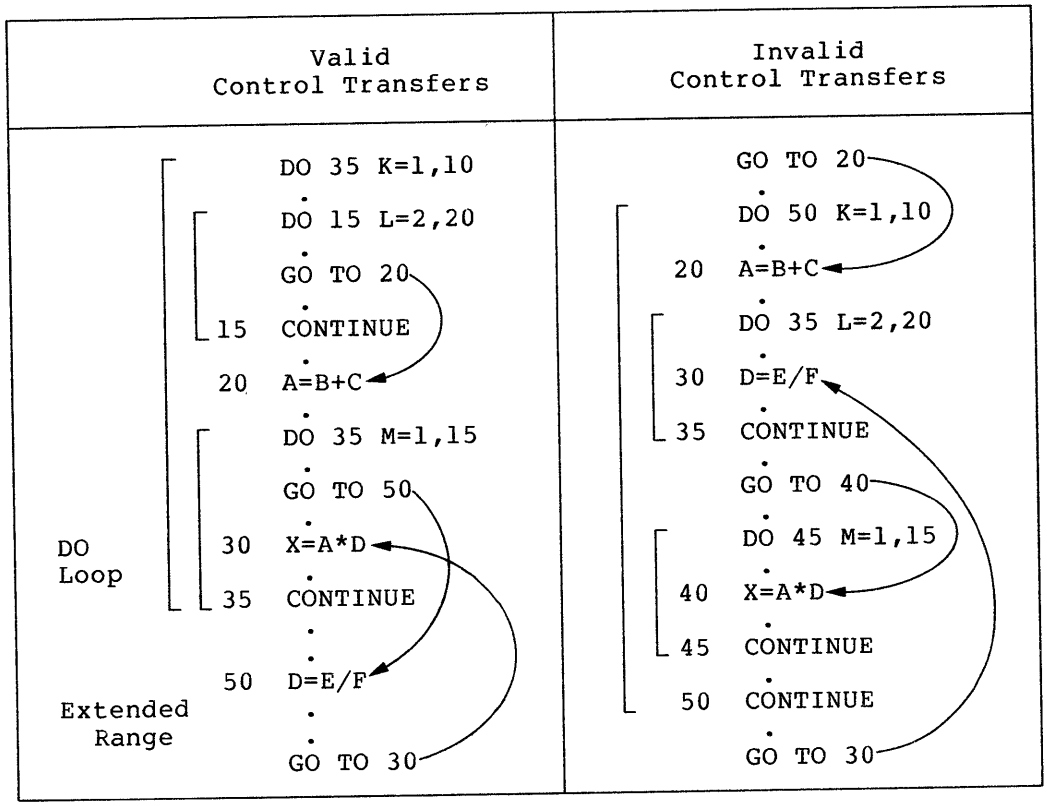


Figure 4-3 Control Transfers and Extended Range

4.4 CONTINUE STATEMENT

The CONTINUE statement transfers control to the next executable statement. It is used primarily as the terminal statement of a 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

4.5 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]]...)]
```

s

The name of a subroutine subprogram or other external procedure; or a dummy argument associated with a subroutine subprogram or other external procedure.

a

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 of the subroutine.

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 4.6.

4.6 RETURN STATEMENT

The RETURN statement is used to transfer 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.

RETURN statement examples:

```
     SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
     DIMENSION DATA(N), PRNT(N)
     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 0, a normal return is taken, and the calling program continues at the first executable statement following CALL CHECK. If Z equals 0, however, the first alternate return (RETURN 1) is taken; and if Z is greater than 0, 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 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.

4.7 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]
```

disp
A character constant or a decimal digit string of 1 to 5 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 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

4.8 STOP STATEMENT

The STOP statement terminates program execution.

The STOP statement has the form:

```
STOP  [disp]
```

disp
A character constant or a decimal digit string of 1 to 5 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.

Examples of STOP statements are:

```
STOP 98

STOP 'END OF RUN'
```

4.9 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
- EXTERNAL statement -- defines the specified symbolic names as external procedure names
- DATA statement -- assigns initial values to variables, arrays, and array elements before program execution
- PARAMETER statement -- assigns a symbolic name to a constant value
- 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 use.

IMPLICIT

5.1 IMPLICIT STATEMENT

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

The IMPLICIT statement has the form:

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

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

a
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.

You cannot label IMPLICIT statements.

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, and character type declarations.

The following rules apply to type declaration statements:

- Type declaration statements must precede all executable statements
- You can declare the data type of a symbolic name only once.
- You cannot label type declaration statements.

5.2.1 Numeric Type Declaration Statements

Numeric type declaration statements have the form:

```
typ v[,v]...
```

typ
Any data type specifier except CHARACTER.

v
The symbolic name of a variable, array, statement function or function subprogram, or an array declarator.

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.

Examples of numeric type declaration statements are:

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

INTEGER*2 I, J, K, M12*4, Q, IVEC*4(10)
REAL*8 WX1, WXZ, WX3*4, WX5, WX6*8
```

5.2.2 Character Type Declaration Statements

Character type declaration statements have the form:

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

v
The symbolic name of a variable, array, function subprogram, or an array declarator.

len
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.

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 a passed length. Only a dummy argument or function name can have this length specification (see Chapter 6).

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 0 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.

Examples of character type declaration statements follow.

```
CHARACTER*32 NAMES(100), SOCSEC(100)*9, NAMETY
```

This statement specifies an array NAMES comprising 100 32-character elements, an array SOCSEC comprising 100 9-character elements, and a variable NAMETY, 32 characters long.

```
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 5.8.)

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

This statement specifies an array LETTER comprising 26 one-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.

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)]...
```

a(d)
An array declarator (see Section 2.5.1).

a
The symbolic name of an array.

d
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. 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 other versions of FORTRAN.

For further information on arrays and on storing 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.

You cannot label DIMENSION statements.

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)
```

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 order of variables and arrays in each common block.

The COMMON statement has the form:

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

cb
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
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, if one program unit contains the statement:

```
COMMON CENTS
```

and another program unit contains the statement:

```
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 high-order 2 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 puts 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)]...
```

nlist
A list of variables, array elements, arrays, and 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 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 corresponding 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 seven-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

The following statements also align 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

The following statements also align 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.

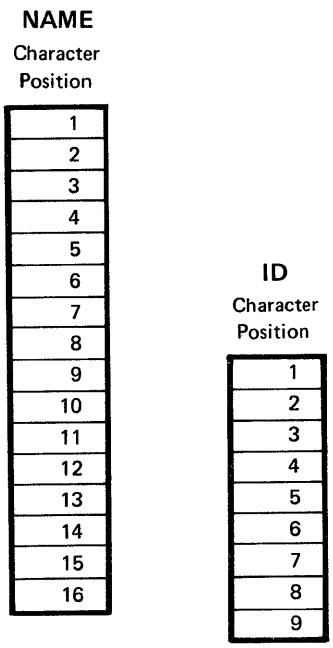


Figure 5-3 Equivalence of Substrings

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.

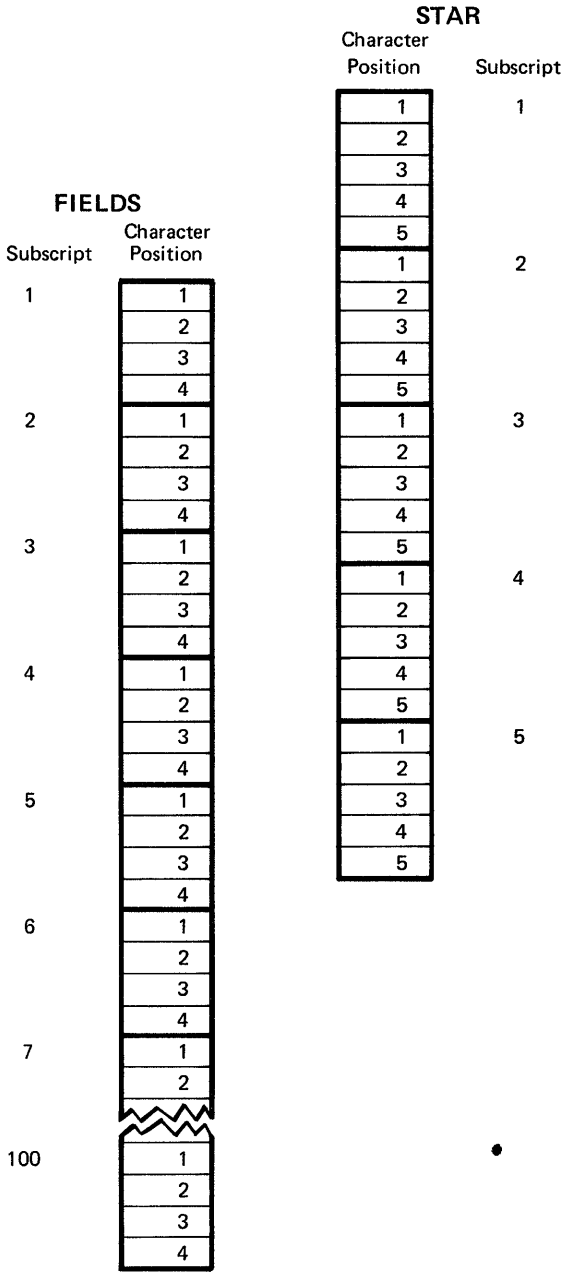


Figure 5-4 Equivalence of Character Arrays

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.

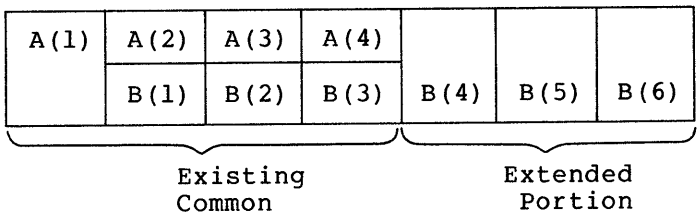
You also cannot use the EQUIVALENCE statement to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

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:

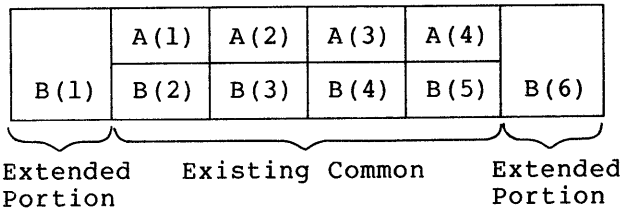
Valid

```
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.

EXTERNAL

5.6 EXTERNAL STATEMENT

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

An external procedure can be a user-supplied function or subroutine subprogram or a FORTRAN library function.

The EXTERNAL statement has the form:

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

v
The symbolic name of a subprogram or the name of a dummy argument associated with a subprogram name.

The EXTERNAL statement declares that each name in the list is an external procedure name. Such a name can then appear as an actual argument to a subprogram; the subprogram can use the associated dummy argument name in a function reference or CALL statement.

If an asterisk (*) precedes a name in the list, the name identifies a user-supplied function or subprogram, not a FORTRAN library function. Use the asterisk only when a user-supplied function or subprogram has the same name as a FORTRAN library function. (See Section 6.3 for additional information on FORTRAN library functions.)

Note, however, that a complete function reference used as an argument (such as, CALL SUBR(A,SQRT(B),C)) represents a value, not a subprogram name. The function name need not be defined in an EXTERNAL statement.

An example of the 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.

DATA

5.7 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/]...
```

nlist
A list of one or more variable names, array names, array element names or character substring names, separated by commas. Subscript expressions and expressions in substring references must be integer constant expressions.

clist
A list of constants; clist constants have one of the following forms:

```
value

n * value
```

n
Used when you specify clist as n * value. Specifies the number of times the same value is to be assigned to successive entities in the associated nlist. The value of 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 that is longer than 4 bytes, the leftmost digits are initialized to zero. If the variable or array element is less than 4 bytes, 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 1 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 0 to all ten 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, FF.

PARAMETER

5.8 PARAMETER STATEMENT

The PARAMETER statement assigns a symbolic name to a constant.

The PARAMETER statement has the form:

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

p
A symbolic name.

c
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); c can be any valid FORTRAN constant.

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

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

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.

Examples

```
PARAMETER  PI=3.1415927, DPI=3.14159265358979323 8D0
PARAMETER  PIOV2=PI/2, DPIOV2=DPI/2
PARAMETER  FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

PROGRAM

5.9 PROGRAM STATEMENT

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

The PROGRAM statement has the form:

```
PROGRAM nam
```

nam
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. It also must not be the same as the name of any subprogram, entry, or common block in the same executable program.

BLOCK DATA

5.10 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]
```

nam
A symbolic name.

You can use type declaration, IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, 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 must not have a statement label.

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-written, or supplied as part of the VAX-11 FORTRAN IV-PLUS system. User-written subprograms include:

- Arithmetic statement functions
- Functions
- Subroutines

Subprograms supplied with the FORTRAN system include:

- Processor-defined functions
- Generic functions
- Character functions

Generally the program that invokes 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 any value assigned to the dummy argument is also assigned to the corresponding actual argument.

6.1.1 Actual Argument and Dummy Argument Association

There must be agreement between actual arguments and dummy arguments. 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 different sized arrays 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 6.1.1.2.)

Section 6.1.1.3 describes the use of character constants and Hollerith constants as actual arguments.

Section 6.1.1.4 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)
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.

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 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,J)
DIMENSION A(I)
A(I) = J
RETURN
ENTRY S1 (I,A,K,L)
A(I) = A(I) + 1
RETURN
END
```

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

```
DIMENSION B(10)
```

The following statement sets B(2)=3:

```
CALL S(B,2,3)
```

The next statement increments B(5) by 1:

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

6.1.1.2 **Passed Length Character Arguments** - A passed length character argument must be a dummy argument. A passed length character argument 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 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 = 1, 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))
```

6.1.1.3 **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 IV-PLUS, 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 only correspond 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, while the actual argument 6HSTRING in the call to HOLLSUB must correspond to a Hollerith dummy argument.

6.1.1.4 **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 a 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. There are two kinds of built-in functions:

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

6.1.2.1 **Argument List Built-In Functions** - To call subprograms written in languages other than FORTRAN (such as system services), you may need to pass the actual arguments in a different form than FORTRAN uses. 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 must not 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 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 IV-PLUS 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
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)
```

v
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 IV-PLUS 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 perform 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 having to duplicate 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
Arithmetic statement function	Arithmetic statement function definition	Function reference
Function	FUNCTION ENTRY RETURN	Function reference
Subroutine	SUBROUTINE ENTRY RETURN	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 Arithmetic Statement Functions

An arithmetic 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 arithmetic statement function, the computation is performed. The resulting value is made available to the expression that contains the arithmetic statement function reference.

The arithmetic statement function definition statement has the form:

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

f
The name of the arithmetic statement function.

p
A dummy argument.

e
An expression.

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

An arithmetic statement function reference has the form:

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

f
The name of the function.

p
An actual argument.

When an arithmetic statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the arithmetic 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 an arithmetic 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 numeric data types, but cannot be 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 arithmetic 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 arithmetic statement function to represent any other entity within the same program unit.

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

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

An arithmetic 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 arithmetic statement function definitions follow.

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

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

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 arithmetic 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 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 **Numeric Functions** - The FUNCTION statement has the form:

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

typ
One of the numeric data type specifiers.

nam
The name of the function.

m
Unsigned, nonzero integer constant specifying the length of the data type.

p
A dummy argument.

6.2.2.2 **Character Functions** - The CHARACTER FUNCTION statement has the form:

```
CHARACTER[*n] FUNCTION nam [([p[,p]...])]
```

n
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.

nam
The name of the function.

p
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]...])
```

nam
The symbolic name of the function.

p
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 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. It must not have a statement label. A function subprogram cannot contain a SUBROUTINE statement, a BLOCK DATA 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:

$$Xi+1 = Xi - \left[\frac{\cosh(Xi)+\cos(Xi)-A}{\sinh(Xi)-\sin(Xi)}\right]$$

This is calculated repeatedly until the difference between Xi and Xi+1 is less than 1.0E-6. The function uses the FORTRAN library 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 executing 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 statement to return control to the calling program unit.

The SUBROUTINE statement has the form:

```
SUBROUTINE nam [([p[,p]...])]
```

nam
The name of the subroutine.

p
A dummy argument. You can specify a dummy argument as an alternate return argument by placing an asterisk in the argument list.

Section 4.5 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. It must not have a statement label.

A subroutine subprogram cannot contain a FUNCTION statement, a BLOCK DATA 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 0, the normal return is taken; if equal to 0, return is to statement label 10 in the main program; if more than 0, 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]...])]
```

nam
The entry name.

p
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. There is no retention of argument association 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 either numeric data type or character data type. 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, the entry name must also be of character data type and must have the same length specification as 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 6.1.1.2).

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 LIBRARY FUNCTIONS

FORTRAN library functions are provided to perform commonly used mathematical computations.

The FORTRAN library functions are listed in Appendix B. Function references to FORTRAN library 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 B gives the data type of each library function and of the actual arguments. Section 6.3.4 also describes the character functions.

6.3.1 Processor-Defined Function References

The FORTRAN library function names are called processor-defined function names. Note that the processor-defined functions include both the Intrinsic Functions and the Basic External Functions defined in ANS FORTRAN.

Normally, a name in the table of processor-defined function names refers to the FORTRAN library function with that name. However, the name can refer to a user-defined function under any of the following conditions:

- The name appears in a type declaration statement specifying a different data type than shown in the table.
- The name is prefixed with an asterisk and appears in an EXTERNAL statement.
- The name is used in a function reference with arguments of a different data type than shown in the table.

Except when they are used in an EXTERNAL statement and are prefixed by an asterisk, processor-defined 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 a processor-defined function does not change if you use an IMPLICIT statement to change the implied data type rules.

Use of a processor-defined function name in an EXTERNAL statement with an asterisk prefix specifies that the name refers to a function or subroutine that you will provide as an external subprogram.

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 double precision variable, the function reference SIN(D) refers to the double precision 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. These names can be used 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:

- In a type declaration statement
- As the name of an arithmetic statement function
- As a dummy argument name, common block name, variable or array name

Using a generic name in an EXTERNAL 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 programs.

Table 6-3
Generic Function Name Summary

Generic Name	Data Type of Argument	Data Type of Result
ABS	Integer Real Double Complex	Integer Real Double Real
AINT, ANINT	Real Double	Real Double
INT, NINT	Real Double	Integer Integer
SNGL	Integer Double	Real Real
DBLE	Integer Real	Double Double
MOD, MAX, MIN, SIGN, DIM	Integer Real Double	Integer Real Double
EXP, LOG, SIN, COS, SQRT	Real Double Complex	Real Double Complex
LOG10, TAN, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, TANH	Real Double	Real Double

6.3.3 Processor-Defined and Generic Function Usage

Figure 6-2 shows the use of processor-defined and generic function names. In this figure, a single executable program uses the name SIN in four distinct ways:

- As the name of an arithmetic statement function
- As a generic function name
- As a processor-defined 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.

```
C
C   COMPARE WAYS OF COMPUTING SINE.
C
    PROGRAM SINES
    PARAMETER PI = 3.14159265358979238D0
    REAL*8 X
    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   MAKE PROCESSOR-DEFINED FUNCTION SIN EXTERNAL (Note 3)
    EXTERNAL SIN
    COMMON V(3)
C   GENERIC REFERENCE TO DOUBLE PRECISION SINE (Note 4)
    V(1) = SIN(Y)
C   PROCESSOR-DEFINED FUNCTION SINE AS ACTUAL ARGUMENT. (Note 5)
    CALL SUB(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 A, SIN
    COMMON V(3)
C   EVALUATE PROCESSOR-DEFINED 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 An arithmetic 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.

2 The arithmetic statement function SIN is called.

3 The name SIN is declared external so that the single precision, processor-defined 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, processor-defined 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 Library Functions

Character library functions are functions that take character arguments or return character 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)
  ```

  c
  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, it returns the substring's starting position. If c2 occurs more than once in c1, the starting position of the first occurrence is returned. If c2 does not occur in c1, the value zero is returned. The INDEX function has the form:

  ```
  INDEX (c1, c2)
  ```

c1
A character expression specifying the string to be searched for the substring specified by c2.

c2
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)
```

c
The character to be converted to an ASCII code. If c is longer than 1 byte, only the value of the first byte is returned. The remainder are 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)
```

i
An integer expression.

Examples illustrating the LEN and INDEX functions follow.

LEN Function Example:

```
      SUBROUTINE REVERSE(S)
      CHARACTER T, S*(*)

      J = LEN(S)
      IF (J .GT. 1) THEN
          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
          ENDIF

      RETURN
      END
```

INDEX Function Example:

```
      SUBROUTINE FIND_SUBSTRINGS(SUB, S)
      CHARACTER*(*) SUB,S
      CHARACTER*132 MARKS

      I = 1
      K = 1
      MARKS = ' '

   10 J = INDEX(S(I:), SUB)
      IF (J .NE. 0) THEN
          I = I + (J-1)
          MARKS(I:I) = '#'
          K = I
          I = I+1
          IF (I .LE. LEN(S)) GO TO 10
          ENDIF

      WRITE(7,100) S, MARKS(:K)
  100 FORMAT( 2(/1X, A) )
      END
```

CHAPTER 7

INPUT/OUTPUT STATEMENTS

FORTRAN programs use READ and ACCEPT statements for input; and WRITE, TYPE, and PRINT statements for output. Some forms of these statements are used with format specifiers that control the translation and editing of the data to and from internal (binary) form and external character (readable) form.

Each READ or WRITE statement refers to the logical unit to or from which data is to be transferred. A logical unit can be connected to a device or file by the OPEN statement (see Section 9.1).

The ACCEPT, TYPE, and PRINT statements do not refer to logical units; rather, they transfer data between the program and an implicit logical unit. The ACCEPT and TYPE statements are normally connected to the default input or output device and the PRINT statement is normally connected to the default output device.

Input/output (I/O) statements are grouped into the following categories:

- Formatted Sequential I/O -- transmits character data using format specifiers to control translation of data from internal (binary) form to character (readable) form on output, and vice versa on input.
- List-Directed Sequential I/O -- transmits character data. The data type of the I/O list elements controls the translation of data to internal form on input and to character form on output.
- Unformatted Sequential I/O -- transmits data in internal form without translation.
- Formatted Direct Access I/O -- transmits character data to and from direct access files using format specifiers to control translation of data from internal form to character form on output, and vice versa on input.
- Unformatted Direct Access I/O -- transmits binary data without translation to and from direct access files.
- Internal I/O -- ENCODE and DECODE statements translate and transfer data between variables and arrays in the FORTRAN program.

These categories of I/O statements are described in Sections 7.2 through 7.7.

Section 7.1 below describes the general components of I/O statements.

7.1 I/O STATEMENT COMPONENTS

The following sections describe logical unit numbers, direct access record numbers, format specifiers, I/O lists, and transfer of control if an error or end-of-file condition occurs.

7.1.1 Logical Unit Numbers

I/O statements use logical unit numbers to refer to files and I/O devices. A logical unit number is an integer expression with a value in the range 0 through 99. It is converted, if necessary, to integer data type before use.

You can use an OPEN statement (see Chapter 9) to connect a file or I/O device to a logical unit. Chapter 3 of the VAX-11 FORTRAN IV-PLUS User's Guide describes other ways to connect a file or I/O device to a logical unit.

Some I/O statements do not contain an explicit logical unit number. Instead, these statements use a system-specific, implicit logical unit number. The VAX-11 FORTRAN IV-PLUS User's Guide describes the use of implicit logical unit numbers in greater detail.

7.1.2 Direct Access Record Numbers

When using direct access I/O, you must indicate which record is to be read or written. To do so, you specify a direct access record number, which is a numeric expression. If necessary, the record number is converted to integer data type before it is used. The value of a direct access record number must be greater than or equal to 1, and less than or equal to the number of records defined for the file.

For a more general discussion of record numbers, refer to Chapter 3 of the VAX-11 FORTRAN IV-PLUS User's Guide.

7.1.3 Format Specifiers

Format specifiers are used in formatted I/O statements and can be any of the following:

- A label of a FORMAT statement
- A character variable name
- A character array name
- A character array element
- A character substring reference
- A numeric array name which contains a value that can be interpreted as a format

Chapter 8 describes FORMAT statements.

In sequential I/O statements, you can use an asterisk instead of a format specifier to denote list-directed formatting. Section 7.3 describes list-directed I/O.

7.1.4 Input/Output Records

I/O statements transfer all data in terms of records. The amount of data that one record can contain, and the way records are separated, depend on how the data is transferred.

In unformatted I/O, the I/O statement specifies the amount of data to be transferred. In formatted I/O, the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.

Executing an input or output statement initiates transfer of a new record. If an input statement requires only part of a record, the remainder of the record is ignored. Normally, the data transferred by an I/O statement constitutes one record. However, formatted sequential I/O statements can transfer more than one record.

7.1.5 Input/Output Lists

The I/O list in an input or output statement contains the names of variables, arrays, array elements, and character substrings whose values are to be transferred. The I/O list in an output statement can also contain constants and expressions.

An I/O list has the form:

```
s[,s]...
```

s
A simple list or an implied DO list.

The I/O statement assigns input values to, or transfers values from, the list elements in the order in which they appear, from left to right.

7.1.5.1 **Simple Lists** - A simple I/O list element can be a single variable, array, array element, character substring reference, constant, or expression. A simple I/O list consists of either a simple I/O list element or a group of two or more simple I/O list elements separated by commas and enclosed in parentheses.

When you use an unsubscripted array name in an I/O list, a READ or ACCEPT statement reads enough data to fill every element of the array; a WRITE, TYPE, or PRINT 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 a READ or ACCEPT statement, variables in the I/O list can be used in array subscripts or substring references later in the list. For example:

```
      READ (1,1250) J,K,ARRAY(J,K)
1250  FORMAT (I1,X,I1,X,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 in the array subscript.

An output statement I/O list can 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 cannot refer to a function subprogram that performs I/O on the same logical unit.

An input statement I/O list cannot contain an expression, except as a subscript expression in an array reference or as an expression in a substring reference.

7.1.5.2 **Implied DO Lists** - Implied DO lists are used to: specify iteration within an I/O list; transfer part of an array; or transfer array elements in a sequence different than the order of subscript progression. This type of list element functions as though it were a part of an I/O statement in a DO loop; it uses the control variable of the implied DO statement to specify the value(s) to be transferred during each iteration of the loop.

An implied DO list has the form:

```
(list,i=e1,e2[,e3])
```

list
An I/O list.

i
An integer, real, or double precision variable.

e1,e2,e3
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 can refer to the variable i (which is the control variable) as long as it does not change the value of i. The list specifies the range of the implied DO list. For example:

```
WRITE (3,200) (A,B,C, I=1,3)

WRITE (6,15) L,M,(I,(J,P(I),Q(I,J),J=1,L),I=1,M)

READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

The first control variable definition is equivalent to the innermost DO loop of a set of nested loops, and therefore varies most rapidly. For example:

```
      WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
150   FORMAT (F10.2)
```

This statement is similar to:

```
      DO 50 K=1,10,2
      DO 50 L=1,10
      WRITE (6,150) FORM(K,L)
150   FORMAT (F10.2)
50    CONTINUE
```

Because the inner DO loop is executed ten 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 the 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)
```

This statement assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ...  , Q(1,10),
P(2), Q(2,1), Q(2,2), ...  , Q(2,10),
 .      .       .            .
 .      .       .            .
P(5), Q(5,1), Q(5,2), ...  , Q(5,10)
```

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), 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.

7.1.6 Transferring Control on End-of-File or Error Conditions

Any READ or WRITE statement can specify that control is to transfer to another statement if the I/O statement encounters an end-of-file or error condition. The specifications have the following forms, respectively, for end-of-file and error conditions:

```
END=s

ERR=s
```

s

The label of an executable statement to which control is to transfer.

A sequential READ or WRITE statement can contain either or both the above specifications, in any order. The specification(s) must follow the unit number, record number, and/or format specification.

An end-of-file condition occurs when no more records exist in a sequential file, or when an end-file record produced by the ENDFILE statement is encountered. 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 no END=s specification is present, an error condition occurs.

If a READ or WRITE 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 no ERR=s specification is present, the I/O error terminates program execution.

The statement label in the END=s or ERR=s specification must refer to an executable statement within the same program unit as the I/O statement.

An END= specification in a WRITE statement or a direct access READ statement is ignored. If you attempt to read or write a record using a record number greater than the maximum specified for the logical unit, an error condition occurs.

The VAX-11 FORTRAN IV-PLUS User's Guide describes system subroutines that you can use to control error processing. These subroutines can also obtain information from the I/O system on the type of error that occurred.

Examples of I/O statements follow.

```
READ (8,END=550) (MATRIX(K),K=1,100)
```

This statement transfers control to statement 550 when an end-of-file condition occurs on logical unit 8.

```
WRITE (6,50,ERR=390)
```

This statement transfers control to statement 390 when an error occurs.

```
READ (1,FORM,ERR=150,END=200) ARRAY
```

This statement transfers control to statement 150 when an error occurs and to statement 200 when the end-of-file condition occurs.

7.2 FORMATTED SEQUENTIAL INPUT/OUTPUT

Formatted sequential I/O statements are used with format specifiers to translate output data from internal (binary) form to external character (readable) form; and to translate input data from external to internal form.

READ
ACCEPT

7.2.1 Formatted Sequential Input Statements

A formatted sequential READ statement transfers data from the specified logical unit. If a formatted sequential READ statement does not have a logical unit number, it uses an implicit logical unit.

The formatted sequential ACCEPT statement is similar to a formatted sequential READ statement except that it always uses an implicit logical unit number.

Formatted sequential input statements have the forms:

```
READ  (u,f[,END=s][,ERR=s])[list]

READ  f[,list]

ACCEPT  f[,list]
```

u
A logical unit number.

f
A format specifier.

s
The label of an executable statement.

list
An I/O list.

A statement of the following form causes data to be read from a system-defined logical unit:

```
READ 200, ALPHA,BETA,GAMMA
```

Characters transferred by formatted sequential statements are translated to the internal form specified by the format specifier. The resulting values are assigned to the elements of the I/O list.

If the number of list elements is less than the number of input record fields, the excess portion of the record is ignored.

Usually a single formatted record is transferred by the execution of a formatted sequential input statement. However, the format specifier can specify that more than one record is to be read during execution of a single input statement.

If the FORMAT statement associated with a formatted input statement contains a Hollerith or character constant, input data is read and stored directly into the format specification.

If no I/O list is present, data transfer occurs only between the record and the format specifier. For example:

```
      READ (5,100)
100   FORMAT (15HΔDATAΔGOESΔHERE)
```

These statements read 15 characters from the next record on logical unit 5. If the 15 characters are:

```
REVIEW SECTIONS
```

The FORMAT statement becomes:

```
100   FORMAT (15HREVIEWΔSECTIONS)
```

Examples of formatted sequential input statements follow.

```
      READ (1,300) ARRAY
300   FORMAT (20F8.2)
```

These statements read a record from logical unit 1, and assign fields to ARRAY.

```
      READ (5,50) CHARV
50    FORMAT (A25)
```

These statements read a record from logical unit 5, and assign a character field to character variable CHARV.

```
      READ 100, ICOUNT,ALPHA,BETA
100   FORMAT (I5, F8.2, F5.1)
```

These statements read a record from an implicit logical unit, and assign fields to integer variable ICOUNT and real variables ALPHA and BETA.

```
      CHARACTER*10 CHARAR(5)
      ACCEPT 200, CHARAR
200   FORMAT (5A10)
```

These statements read a record from an implicit logical unit, and assign fields to the character array CHARAR.

WRITE
TYPE
PRINT

7.2.2 Formatted Sequential Output Statements

The formatted sequential WRITE statement transfers data to the specified logical unit.

The formatted sequential TYPE and PRINT statements are similar to the formatted sequential WRITE statement, except that output is directed to an implicit logical unit.

The formatted sequential output statements have the form:

```
WRITE  (u,f[,ERR=s])[list]

TYPE  f[,list]

PRINT  f[,list]
```

u
A logical unit number.

f
A format specifier.

s
The label of an executable statement.

list
An I/O list.

The I/O list specifies a sequence of values that are converted to characters and positioned as specified by the format specifier. If no I/O list is present, data transfer occurs only between the format specifier and the record.

The data transferred by a formatted sequential output statement normally constitutes one formatted record. However, the format specifier can specify that additional records are to be written during execution of a single output statement.

Numeric data output under format control is rounded during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. To avoid loss of precision, use unformatted output.)

The records transmitted by a formatted WRITE statement must not exceed the length that the specified device can accept. For example, a line printer typically cannot print a record longer than 132 characters.

Examples of formatted sequential output statements follow.

```
      WRITE (6, 650)
650   FORMAT ('ΔHELLOΔTHERE')
```

These statements write the contents of the FORMAT statement to logical unit 6.

```
      CHARACTER*16 NAME,JOB
      PRINT 400, NAME, JOB

400   FORMAT ('NAME=',A,'JOB=',A)
```

These statements write one record consisting of four fields.

```
      WRITE (1,95) AYE,BEE,CEE
95    FORMAT (3F8.5)
```

These statements write one record, consisting of three fields, to logical unit 1.

```
      WRITE (1,950) AYE,BEE,CEE
  950 FORMAT (F8.5)
```

These statements write three separate records, consisting of one field each, to logical unit 1.

In the last example, format control reaches the rightmost parenthesis of the FORMAT statement before all elements of the I/O list are output. Each time this occurs, the current record is terminated and a new record is initiated. Thus, three separate records are written.

7.3 LIST-DIRECTED SEQUENTIAL INPUT/OUTPUT

List-directed sequential I/O statements provide a way to obtain simple formatted sequential input or output without using FORMAT statements.

On input, values are read from the logical unit, converted to internal form, and assigned to the elements of the I/O list. On output, values in the I/O list are converted to character form and written in a fixed format according to the data type of the value. The I/O list is required.

Records written by list-directed output statements can be input by list-directed input statements if they contain only numeric and logical values.

Both formatted and list-directed sequential I/O statements can refer to the same logical unit. All operations permitted for formatted sequential I/O are permitted for list-directed I/O. However, backspacing over list-directed records leaves the file position undefined. When files are read that contain both formatted and list-directed records, the user program should ensure that each record is read with the same kind of formatting that was used to write it.

READ
ACCEPT

7.3.1 List-Directed Input Statements

A list-directed READ statement transfers data from the specified logical unit, translates data from external to internal form, and assigns the input values to the elements of the I/O list in the order in which they appear, from left to right. If a list-directed READ statement does not have a logical unit number, it uses an implicit logical unit number.

The list-directed ACCEPT statement is similar to a list-directed READ statement except that it always uses an implicit logical unit number.

The list-directed input statements have the forms:

```
READ (u,*[,END=s][,ERR=s]) list

READ *,list

ACCEPT *,list
```

u
A logical unit number.

*****
Indicates list-directed formatting.

s
The label of an executable statement.

list
An I/O list.

The external record contains a sequence of values and value separators. A value can be:

- A constant
- A null value
- A repetition of constants in the form r*c
- A repetition of null values in the form r*

The following paragraphs describe these values.

Each input 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 is either T (true) or F (false). A character constant is delimited by apostrophes; any apostrophe in the character constant is 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. Spaces can occur between the commas. A null value specifies that the corresponding list element remains unchanged. A null value cannot be used for either part of a complex constant, but can represent an entire 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 can be:

- One or more spaces or tabs
- A comma, with or without surrounding spaces or tabs
- A slash, with or without surrounding spaces or tabs -- terminates processing on the input statement and record; all remaining I/O list elements are 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, double precision, 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 list-directed input statements follows.

The 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
     .
     .
     .
```

The record read contains:

```
4 6.3 (3.4,4.2), (3, 2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

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 will be unchanged.

WRITE
TYPE
PRINT

7.3.2 List-Directed Output Statements

The list-directed WRITE statement transfers the elements in the I/O list to the specified unit, translating and editing each value according to the data type of the value.

The list-directed TYPE and PRINT statements are similar to the list-directed WRITE statement, except that output is directed to an implicit logical unit.

The list-directed output statements have the forms:

```
WRITE (u,*[,ERR=s]) list

TYPE  *,list

PRINT *,list
```

u
A logical unit number.

*****
Indicates list-directed formatting.

s
The label of an executable statement.

list
An I/O list.

Except for character constants, the output values have the same forms as the input constant values described above in Section 7.3.1. Character constants are transferred without delimiting apostrophes; each internal apostrophe is represented by only one apostrophe. Consequently, records containing list-directed character output can be printed but cannot be used for list-directed input.

Table 7-1 lists the default output formats for each data type.

Table 7-1
List-Directed Output Formats

Data Type	Output Format
LOGICAL*1	I5
LOGICAL*2	L2
LOGICAL*4	L2
INTEGER*2	I7
INTEGER*4	I12
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	1X,'(',1PG14.7, ',', 1PG14.7,')'
CHARACTER	1X,An (n is the length of the character expression)

The list-directed output statements do not produce octal values, null values, slash separators, or repeated forms of values. Each output record begins with a space for carriage control. Each output statement writes one or more complete records. Each output value is contained within a single record, except for character constants that are longer than a record.

An example of list-directed output statements follows.

```
PRINT *,'THEΔARRAYΔZΔIS',Z
TYPE *,'THEΔANSWERΔIS',(I,XX(I),I=1,10)
```

If a program unit consists of:

```
DIMENSION A(5)
DATA A/5*3.4/
WRITE (1,*) 'ARRAYΔVALUESΔFOLLOW'
WRITE (1,*) A,5
```

This program unit writes the following records:

```
ARRAYΔVALUESΔFOLLOW
ΔΔΔ3.400000ΔΔΔΔΔΔΔ3.400000ΔΔΔΔΔΔΔ3.400000ΔΔΔΔΔΔΔ3.400000
ΔΔΔ3.400000ΔΔΔΔΔΔΔΔΔΔΔ5
```

7.4 UNFORMATTED SEQUENTIAL INPUT/OUTPUT

Unformatted sequential I/O transfers data in internal (binary) format. That is, no conversion or editing takes place. Unformatted I/O is generally used when data output by a program will be subsequently input by the same (or a similar) program. 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.

READ

7.4.1 Unformatted Sequential Input Statement

The unformatted sequential READ statement inputs one unformatted record from the specified logical unit, and assigns the untranslated fields of the record to the I/O list elements in the order in which they appear, from left to right. The data type of each element determines the amount of data that element receives.

The unformatted sequential READ statement has the form:

```
READ (u[,END=s][,ERR=s])[list]
```

u
A logical unit number.

s
The label of an executable statement.

list
An I/O list.

An unformatted sequential READ statement reads exactly one record. If the I/O list does not use all the values in the 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 an unformatted sequential READ 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 be used to read records created by unformatted sequential WRITE statements.

Examples of unformatted sequential input statements follow.

```
READ (1) FIELD1, FIELD2
```

This statement reads one record from logical unit 1 and assigns values to variables FIELD1 and FIELD2.

```
READ (8)
```

This statement advances logical unit 8 one record.

WRITE

7.4.2 Unformatted Sequential Output Statement

The unformatted sequential WRITE statement transfers the untranslated values of the elements in the I/O list to the specified logical unit. That is, one unformatted record is output.

The unformatted sequential WRITE statement has the form:

```
WRITE (u[,ERR=s])[list]
```

u
A logical unit number.

s
The label of an executable statement.

list
An I/O list.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

Examples of unformatted sequential output statements follow.

```
WRITE (1) (LIST(K),K=1,5)
```

This statement outputs the contents of elements 1 through 5 of array LIST to logical unit 1.

```
WRITE (4)
```

This statement writes a null record on logical unit 4.

7.5 FORMATTED DIRECT ACCESS INPUT/OUTPUT

Formatted direct access I/O transfers character data to and from a file on a direct access device. The OPEN statement (see Section 9.1) establishes the attributes of the direct access file. Each READ or WRITE statement contains a direct access record number.

READ

7.5.1 Formatted Direct Access Input Statement

The formatted direct access READ statement transfers the specified record from the direct access file currently connected to the specified unit. The characters in the record are translated to internal form according to the format specifier. The resulting values are assigned to the elements of the I/O list.

The formatted direct access READ statement has the form:

```
READ (u'r,f[,ERR=s])[list]
```

u
A logical unit number.

r
The direct access record number.

f
A format specifier.

s
The label of an executable statement.

list
An I/O list.

If the I/O list and format specify more characters than a record contains, or specify additional records, an error occurs.

WRITE

7.5.2 Formatted Direct Access Output Statement

The formatted direct access WRITE statement transfers the specified record to the direct access file currently connected to the unit. The list specifies a sequence of values that are translated to characters and positioned as specified by the format specifier.

The formatted direct access WRITE statement has the form:

```
WRITE (u'r,f[,ERR=s])[list]
```

u
A logical unit number.

r
The direct access record number.

f
A format specifier.

s
The label of an executable statement.

list
An I/O list.

If the values specified by the list and format do not fill the record, the unused portion of the record is filled with space characters.

If the I/O list and format specify more characters than can fit into the record, or specify additional records, an error occurs.

7.6 UNFORMATTED DIRECT ACCESS INPUT/OUTPUT

Unformatted direct access I/O transfers data in internal (binary) form to and from a direct access file. The OPEN or DEFINE FILE statement (see Sections 9.1 and 9.7) establishes the attributes of the file. Each direct access READ or WRITE statement contains a direct access record number.

READ

7.6.1 Unformatted Direct Access Input Statement

The unformatted direct access READ statement transfers the specified record from the direct access file currently connected to the specified unit, and assigns the untranslated fields of the record to the I/O list elements.

The unformatted direct access READ statement has the form:

```
READ (u'r[,ERR=s]) [list]
```

u
A logical unit number.

r
The direct access record number.

s
The label of an executable statement.

list
An I/O list.

If the I/O list does not use all the fields in the record (that is, there are more fields 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 record fields, an error occurs.

Examples of unformatted direct access input statements follow.

```
READ (1'10) LIST(1),LIST(8)
```

This statement reads record 10 of a file on logical unit 1, and assigns two integer values to specified elements of array LIST.

```
READ (4'58) (RHO(N),N=1,5)
```

This statement reads record 58 of a file on logical unit 4, and assigns five real values to array RHO.

7.6.2 Unformatted Direct Access Output Statement

The unformatted direct access WRITE statement transfers the untranslated values of the elements in the I/O list to the specified record of the direct access file currently connected to the specified unit.

The unformatted direct access WRITE statement has the form:

```
WRITE (u'r[,ERR=s]) [list]
```

u
A logical unit number.

r
The direct access record number.

s
An executable statement label.

list
An I/O list.

If the values specified by the list do not fill the record, the unused portion of the record is filled with zeros.

If the list specifies more data than can fit into the record, an error occurs.

Examples of unformatted direct access output statements follow.

```
WRITE (2'35) (NUM(K),K=1,10)
```

This statement outputs ten integer values to record 35 of the file connected to logical unit 2.

```
WRITE (3'J) ARRAY
```

This statement outputs the entire contents of ARRAY to the record indicated by the value of J in the file connected to logical unit 3.

ENCODE
DECODE

7.7 ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements transfer data according to format specifiers, translating the data from internal to character form, and vice versa. Unlike conventional formatted I/O statements, however, these data transfers take place entirely between variables or arrays in the FORTRAN program.

The ENCODE and DECODE statements have the forms:

```
ENCODE(c,f,b[,ERR=s])[list]

DECODE(c,f,b[,ERR=s])[list]
```

c
: 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
: A format specifier. If more than one record is specified, an error occurs.

b
: 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.

s
: The label of an executable statement.

list
: 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, similar to 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, similar to 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 2 characters per element, so 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 times 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'
```

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 ($q_1 f_1 s_1 f_2 s_2 \ldots f_n q_n$)

q
Zero or more slash (/) record terminators.

f
A field descriptor or a group of field descriptors enclosed in parentheses.

s
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 the form:

[r]cw[.d]

r
The repeat count for the field descriptor. If you omit r, it is assumed to be 1.

c
A format code (I,O,Z,F,E,D,G,L,A,H,X,T,P,Q,$, or :).

w
The external field width.

d
The number of characters to the right of the decimal point.

The terms r, w, and d must all be unsigned integer constants; r and w must be less than or equal to 32767, and d must be less than or equal to 255. The r term is optional; however, you cannot use it in some field descriptors (see Section 8.1.18). The d term is required in some field descriptors and is invalid in others. You are not allowed to use PARAMETER constants for the terms r, w, or d.

The field descriptors are:

- Integer -- Iw, Ow, Zw
- Logical -- Lw
- Real, double precision, and complex -- Fw.d, Ew.d, Dw.d, Gw.d
- Character -- Aw
- Editing, and character and Hollerith constants -- nH, '...', nX, Tn, nP, Q, $, : (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 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.

The following sections describe each of the field descriptors in detail. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeros.

8.1.1 I Field Descriptor

The I field descriptor transfers decimal integer values. It has the form:

```
Iw
```

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, as a decimal value, 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 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. Blanks following the first non-blank character are treated as zeros.

Input Example:

Format	External Field	Internal Representation
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	312
I4	2ΔΔ8	2008

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. Plus signs, however, are suppressed.

Output Example:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.812	Not permitted: error

8.1.2 O Field Descriptor

The O field descriptor transfers octal values. It has the form:

Ow

The corresponding I/O list element must be of either integer or logical data type.

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. Blanks following the first non-blank character are treated as zeros. 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 Representation
O5	32767	32767
O4	16234	1623
O6	13ΔΔΔΔ	130000
O3	97Δ	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 its octal (2's complement) 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.

Output Example:

Format	Internal (Decimal) Value	External Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O5	13.52	Not permitted: error

8.1.3 Z Field Descriptor

The Z field descriptor transfers hexadecimal values. It has the form:

Zw

The corresponding I/O list element must be of either integer or logical data type.

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 through F; it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of zero. Blanks following the first non-blank character are treated as zeros. 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 Representation
Z3	A94	A94
Z5	A23DEF	A23DE
Z7	9AFΔΔΔΔ	9AF0000
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 its hexadecimal (2's complement) 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.

Output Example:

Format	Internal (Decimal) Value	External Representation
Z4	32767	7FFF
Z5	-32767	Δ8001
Z2	16	10
Z3	25.2	Not permitted: error

8.1.4 F Field Descriptor

The F field descriptor transfers real or double precision values. It has the form:

Fw.d

The corresponding I/O list element must be of either real or double precision 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 or double precision 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. In all F field descriptors, w must be greater than or equal to d+1.

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. 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 (as described in Section 2.3.2 for real constants or Section 2.3.3 for double precision constants), 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 Representation
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 suppressed), 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	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.1	51.44	**
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

8.1.5 E Field Descriptor

The E field descriptor transfers real or double precision values in exponential form. It has the form:

Ew.d

The corresponding I/O list element must be of either real or double precision 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 or double precision value to the corresponding I/O list element. This is exactly the same way that the F field descriptor interprets and assigns data.

Input Example:

Format	External Field	Internal Representation
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E-6	1022.43E-6
E15.3	52.3759663ΔΔΔΔΔ	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 if the value is negative (plus signs are suppressed), a zero, a decimal point, d digits to the right of the decimal points, and a 4-character exponent. The exponent has one of the following two forms:

E+nn

E-nn

nn
A 2-digit integer constant.

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 suppressed), a zero, a decimal point, d digits, and an exponent. Therefore, w must be greater than or equal to d+7.

Output Example:

Format	Internal Value	External Representation
E9.2	475867.222	Δ0.48E+06
E12.5	475867.222	Δ0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****

8.1.6 D Field Descriptor

The D field descriptor transfers real or double precision values in exponential form. It has the form:

Dw.d

The corresponding I/O list element must be of either real or double precision 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 double precision value to the corresponding I/O list element. This is the same way that the F field descriptor interprets and assigns data.

Input Example:

Format	External Field	Internal Representation
D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	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	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	*********

8.1.7 G Field Descriptor

The G field descriptor transfers real or double precision values in a form that, in effect, combines the F and E field descriptors. It has the form:

Gw.d

The corresponding I/O list element must be of either real or double precision 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 or double precision value to the corresponding I/O list element. This is the same way the F field descriptor interprets and assigns data.

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
0.1 ≤ m < 1.0	F(w-4).d, 'ΔΔΔΔ'
1.0 ≤ m < 10.0	F(w-4).(d-1), 'ΔΔΔΔ'
.	.
.	.
.	.
10d-2 ≤ m < 10d-1	F(w-4).1, 'ΔΔΔΔ'
10d-1 ≤ m < 10d	F(w-4).0, 'ΔΔΔΔ'
m ≥ 10d	Ew.d

The 'ΔΔΔΔ' field descriptor, which is, in effect, inserted by the G field descriptor for values within its range, specifies that four 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 suppressed), at least one digit to the left of the decimal point, the decimal point, d digits to the right of the decimal point, and (for values outside the effective range of the G field descriptor) a 4-character exponent. Therefore, w must be greater than or equal to d plus 7.

Output Example:

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0.123457E-01
G13.6	-0.12345678	-0.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ
G13.6	12.34567890	ΔΔ12.3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123.457ΔΔΔΔ
G13.6	-1234.56789012	Δ-1234.57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457.ΔΔΔΔ
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	ΔΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ-0.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.456789
F13.6	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

8.1.8 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 character of the field is the letter T, the value .TRUE. is assigned to the corresponding I/O list element. If the first nonblank character of the field is the letter F, or if the entire field is blank, the value .FALSE. is assigned. Any other value in the external field produces an error.

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.	ΔΔΔΔT
L1	.FALSE.	F

8.1.9 A Field Descriptor

The A field descriptor transfers character or Hollerith values. It has the form:

Aw

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	8
DOUBLE PRECISION	8
COMPLEX	8

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△#△△	(DOUBLE PRECISION)

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, 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	ΔOHMS
A5	VOLTSΔΔΔΔ	VOLTS
A5	AMPERESΔ	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.

8.1.10 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:

$$nHc_1c_2c_3 \ldots c_n$$

n
The number of characters to be transferred.

c
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.

An example of H field descriptor usage follows.

```
      TYPE  100
100   FORMAT (41HΔENTERΔPROGRAMΔTITLE,ΔUPΔTOΔ20ΔCHARACTERS)
      ACCEPT  200
200   FORMAT (20HΔΔTITLEΔGOESΔHEREΔΔΔ)
```

The TYPE statement transfers the characters from the H field descriptor in statement 100 to the user's terminal. The ACCEPT statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the words TITLE GOES HERE. If the user enters less than 20 characters, the remainder of the H field descriptor is filled with spaces to the right.

8.1.10.1 **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''SΔDATEΔIS:Δ',I2,'/',I2,'/',I2)
```

A pair of apostrophes used this way is considered a single character.

8.1.11 X Field Descriptor

The X field 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 transfers n spaces to the external record. For example:

```
      WRITE (6,90) NPAGE
90    FORMAT (13H1PAGEΔNUMBERΔ,I2,16X,23HGRAPHICΔANALYSIS,ΔCONT.)
```

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.

8.1.12 T Field Descriptor

The T field descriptor is a 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, but not greater than the number of characters allowed in the external record.

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:

```
ABCΔΔΔXYZ
```

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 to devices other than the line printer or terminal, the T field descriptor specifies that subsequent data transfer is to begin at the nth character position of the external record. On output to a printer, data transfer begins at position (n-1). The first position of a printed record is reserved for a carriage control character, which is never printed (see Section 8.2). For example:

```
      PRINT 25
25    FORMAT (T51,'COLUMNΔ2',T21,'COLUMNΔ1')
```

These statements would print the following line:

8.1.13 Q Field Descriptor

The Q field 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 field descriptor has no effect except that the corresponding I/O list element is skipped.

8.1.14 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 ('ΔENTERΔRADIUSΔVALUEΔ',$)
      ACCEPT 200
200   FORMAT (F6.2)
```

produce a message on the terminal in the form:

```
ENTERΔRADIUSΔVALUE
```

Your response (for example, 12.) can then go on the same line, as:

```
ENTER RADIUS VALUE 12.
```

8.1.15 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('ΔI=',I2, 'ΔJ=',I2)
2     FORMAT('ΔK=',I2,:,'ΔL=',I2)
```

These statements print the following two lines:

```
I=Δ3ΔJ=
K=Δ4
```

Section 8.6 describes format control in detail.

8.1.16 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.d, Dw.d, or Gw.d.

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 Representation
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 12345.678

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	Δ2.35472 3.45673
E9.2,'Δ,Δ',E5.3	47587.222, 56.123	Δ0.48E+06Δ,Δ*****

8.1.17 Scale Factor

The scale factor lets you alter, during input or output, the location of the decimal point in real and double precision values, and in the two parts of complex values.

The scale factor has the form:

```
nP
```

n

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 field descriptors that are 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 Representation
3PE10.5	ΔΔΔ37.614	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	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 and double precision 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.

8.1.18 Repeat Counts and Group Repeat Counts

You can apply most field descriptors (except H, T, P, or X) to a number of successive data fields by preceding that field descriptor with an unsigned integer constant specifying the number of repetitions. 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))

50    FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
                    1          2          3
```

An H or X 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.

8.1.19 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 one greater than the value of J at the time the format is scanned. The expression is re-evaluated 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 use. 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.

```
      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

8.1.20 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 and d are supplied based on the data type of the I/O list element.

Table 8-2 lists the default values for w and d.

Table 8-2
Default Field Descriptor Values

Field Descriptor	List Element	w	d
I, O, Z	INTEGER*2	7	
I, O, Z	INTEGER*4	12	
L	LOGICAL	2	
F, E, G, D	REAL, COMPLEX	15	7
F, E, G, D	DOUBLE PRECISION	25	16
A	LOGICAL*1	1	
A	LOGICAL*2,INTEGER*2	2	
A	LOGICAL*4,INTEGER*4	4	
A	REAL, COMPLEX	4	
A	DOUBLE PRECISION	8	
A	CHARACTER*n	n	

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. The FORTRAN I/O system recognizes certain characters as carriage control characters. Table 8-3 lists these characters and their effects.

Table 8-3
Carriage Control Characters

Character	Effect
Δ (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,'HEADINGΔLINE'//T51,'SUBHEADINGΔLINE'//)
```

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, 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, or numeric arrays. 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 the following example, the DATA statement assigns a left parenthesis to the character variable FORCHR, and assigns a right parenthesis and three field descriptors to four character variables for later use. The proper field descriptors are then 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.

```
      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 format control takes 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 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, 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 format control encounters an I, O, Z, F, E, D, G, L, A, or Q field descriptor, it determines whether a corresponding element exists in the I/O list. If one does, format control transfers data (translated, as appropriate, to or from external format) between the record and the list element, then proceeds to the next field descriptor (unless the current one is to repeat). If no corresponding list element remains, format control terminates.

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 or nX, the terms r, w, and n must be unsigned integer constants 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. You must specify d in F, E, D, and G field descriptors even if it is zero; and the field width specification (w) must be greater than or equal to d. The decimal point is also required. You must either specify both w and d, or omit them both.

4. In a field descriptor such as nHc1c2 ... cn, 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 a signed or unsigned integer constant 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 or double precision 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 H, X, T 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. However, this descriptor cannot be H, X, P, T, or a character constant.

8. A format specification in a variable, a character substring reference, an array element, or an array must be constructed the same as a format specification in a FORMAT statement, including the opening and closing parentheses.

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 through 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 or double precision constant. It can contain a decimal point and/or an E or D 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 or double precision 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 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 characters.

3. The first character of a record output to a line printer or terminal is 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
I	Iw	Transfers decimal integer values
O	Ow	Transfers octal integer values
Z	Zw	Transfers hexadecimal integer values
F	Fw.d	Transfers real or double precision values
E	Ew.d	Transfers real or double precision values (E exponent field indicator)
D	Dw.d	Transfers real or double precision values (D exponent field indicator)
G	Gw.d	Transfers real or double precision values: on input, acts like F code; on output, acts like E code or F code.
L	Lw	Transfers logical data: on input, transfers characters; on output, transfers T or F
A	Aw	Transfers character or Hollerith values
H	nHc...c or 'c...c'	Transfers data between an external record and the H field descriptor, or character constant
X	nX	Specifies that n characters are to be skipped (on input) or that n spaces are to be transmitted (on output)
T	Tn	Tabulation specifier
Q	Q	Obtains the number of characters remaining to be transferred in an input record.
$	$	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.

- REWIND, BACKSPACE, and FIND -- 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.

- DEFINE FILE -- describes an unformatted, direct access file and associates the file with a logical unit number.

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]...)
```

par
A keyword specification in one of the following forms:

```
key
key = value
```

key
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
NAME	- file name specification for the file
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
RECORDSIZE	- logical record length
RECORDTYPE	- logical record format

- 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 transfers if an error occurs during execution of the OPEN statement

Table 9-1 lists the values accepted for each keyword.

Table 9-1
OPEN Statement Keyword Values

Keyword	Values*	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'	Access method	'SEQUENTIAL'
ASSOCIATEVARIABLE	v	Next direct access record	
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	'SAVE' or 'KEEP' 'PRINT' 'DELETE'	File disposition at close	'SAVE'
ERR	s	Error transfer label	
EXTENDSIZE	e	File allocation increment	Volume or system default
FORM	'FORMATTED' 'UNFORMATTED'	Format type	Depends on access method
INITIALSIZE	e	File allocation	
MAXREC	e	Direct access record limit	
NAME	c	File name specification	
NOSPANBLOCKS	-	Records do not span blocks	
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'	File structure	'SEQUENTIAL'

*NOTES:

c is a character expression, numeric array name, numeric variable name, or numeric array element name
e is a numeric expression
p is an external function
s is a statement label
v is an integer variable name

(continued on next page)

Table 9-1 (Cont.)
OPEN Statement Keyword Values

Keyword	Values*	Function	Default
READONLY	-	Write protection	
RECORDSIZE	e	Record length	As specified at file creation
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'	Record structure	Direct access - 'FIXED'. Formatted seq. access - 'VARIABLE'. Unformatted seq. access - 'SEGMENTED'
SHARED	-	File sharing allowed	
TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	File status at open	'NEW'
UNIT	e	Logical unit number	
USEROPEN	p	User program option	

*NOTES:

c is a character expression, numeric array name, numeric variable name, or numeric array element name
e is a numeric expression
p is an external function
s is a statement label
v is an integer variable name

Keyword specifications can appear in any order. In most cases, they are optional; and, if not present, default values are provided.

The following examples illustrate four uses of the OPEN statement:

```
OPEN (UNIT=1, ERR=100)
```

This statement creates a new sequential formatted file on unit 1 with the default file name FOR001.DAT.

```
OPEN (UNIT=3, TYPE='SCRATCH', ACCESS='DIRECT',
      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, NAME='MTA0:MYDATA.DAT', BLOCKSIZE=8192,
      TYPE='NEW', ERR=14, RECORDSIZE=1024, RECORDTYPE='FIXED')
```

This statement creates a file on magnetic tape with a large block size for efficient processing.

```
OPEN (UNIT=I, NAME='MTA0:MYDATA.DAT', READONLY, TYPE='OLD',
      RECORDSIZE=1024, RECORDTYPE='FIXED', BLOCKSIZE=8192)
```

This statement opens the file created in the previous example for input.

```
CHARACTER*40 FILENAME
   .
   .
   .

OPEN (UNIT=1,NAME=FILENAME,TYPE='OLD')
```

This statement opens an existing file, using the name specified by the character variable FILENAME.

Sections 9.1.1 through 9.1.21 describe the keywords in detail. As used in these sections, a numeric expression can be any integer, real, or double precision 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

This keyword has the form:

```
ACCESS = acc
```

acc
The character constant 'DIRECT', 'SEQUENTIAL', or 'APPEND'.

ACCESS specifies whether the file is direct or sequential access. If you specify 'DIRECT', the file is accessed directly. If you specify 'SEQUENTIAL', the file is accessed sequentially. 'APPEND' implies sequential access and positioning after the last record of the file. The default is 'SEQUENTIAL'.

9.1.2 ASSOCIATEVARIABLE Keyword

This keyword has the form:

```
ASSOCIATEVARIABLE = asv
```

asv
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. This specifier is ignored for a sequential access file.

9.1.3 BLOCKSIZE Keyword

This keyword has the format:

```
BLOCKSIZE = bks
```

bks
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 IV-PLUS User's Guide for more information.

9.1.4 BUFFERCOUNT Keyword

This keyword has the form:

```
BUFFERCOUNT = bc
```

bc
A numeric expression.

BUFFERCOUNT specifies the number of buffers to be associated with the logical unit for multibuffered I/O. The size of each buffer is determined by the BLOCKSIZE keyword. If you do not specify BUFFERCOUNT, or if you specify zero, the system default is assumed.

9.1.5 CARRIAGECONTROL Keyword

This keyword has the form:

```
CARRIAGECONTROL = cc
```

cc
The character constant '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.

9.1.6 DISPOSE Keyword

This keyword has two forms:

```
DISPOSE = dis
DISP = dis
```

dis
The character constant 'SAVE', 'KEEP', 'PRINT', or 'DELETE'.

DISPOSE determines the disposition of the file connected to the unit when the unit is closed. If you specify 'SAVE' or 'KEEP', the file is retained after the unit is closed; this is the default value. If you specify 'PRINT', the file is submitted to the system line printer spooler and is not deleted. If you specify 'DELETE', the file is deleted. A read-only file cannot be printed or deleted. A scratch file cannot be saved or printed.

9.1.7 ERR Keyword

This keyword has the form:

ERR= s

s
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.

9.1.8 EXTENDSIZE Keyword

This keyword has the form:

EXTENDSIZE = es

es
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.

9.1.9 FORM Keyword

This keyword has the form:

FORM = ft

ft
The character constant '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 files, 'UNFORMATTED' is the default.

9.1.10 INITIALSIZE Keyword

This keyword has the form:

```
INITIALSIZE = insz
```

insz
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.

9.1.11 MAXREC Keyword

This keyword has the form:

```
MAXREC = mr
```

mr
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 is ignored for a sequential access file.

9.1.12 NAME Keyword

This keyword has the form:

```
NAME = fln
```

fln
A character expression, numeric array name, numeric variable name, or numeric array element name.

NAME 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 IV-PLUS 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.

9.1.13 NOSPANBLOCKS Keyword

This keyword 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.

9.1.14 ORGANIZATION Keyword

This keyword has the form:

```
ORGANIZATION = org
```

org
The character constant 'SEQUENTIAL' or 'RELATIVE'.

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.

The internal organization of a file does not limit the access allowed a file. Specifically, both sequential and direct access are allowed on files with either sequential or relative organization. However, you must specify fixed length records for direct access, sequential organization files.

See the VAX-11 FORTRAN IV-PLUS User's Guide for more information on internal file organization.

9.1.15 READONLY Keyword

This keyword has the form:

```
READONLY
```

READONLY specifies that an existing file can be read, but prohibits writing to that file.

9.1.16 RECORDSIZE Keyword

This keyword has the form:

```
RECORDSIZE = rl
```

rl
A numeric expression.

RECORDSIZE specifies the logical record length. If the file contains fixed length records, RECORDSIZE specifies the size of each record. If the file contains variable length records, RECORDSIZE specifies the maximum length for any record. If the records are formatted, the length is the number of characters; if the records are unformatted, the length is the number of numeric storage units (4 bytes). 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 RECORDSIZE when you create files with fixed length records or with relative organization.

9.1.17 RECORDTYPE Keyword

This keyword has the form:

```
RECORDTYPE = typ
```

typ
The character constant '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 organization	'FIXED'
Direct access files	'FIXED'
Formatted sequential access files	'VARIABLE'
Unformatted sequential access files	'SEGMENTED'

Segmented records 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).

9.1.18 SHARED Keyword

This keyword 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 IV-PLUS User's Guide for additional information on this keyword.

9.1.19 TYPE Keyword

This keyword has the form:

```
TYPE = typ
```

typ
The character constant 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'.

TYPE specifies the status of 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 'NEW'.

9.1.20 UNIT Keyword

This keyword has the form:

```
UNIT = u
```

u
A numeric expression.

UNIT specifies the logical unit to which a file is to be connected. The unit specification must appear in the list. Another file cannot be connected to the logical unit when the OPEN statement is executed.

9.1.21 USEROPEN Keyword

The keyword has the form:

```
USEROPEN = p
```

p
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 which are not directly available from FORTRAN, while retaining the convenience of writing programs in FORTRAN. See the VAX-11 Common Run-Time Procedure Library Reference Manual for more information on USEROPEN.

CLOSE

9.2 CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit. It has the form:

$$\text{CLOSE (UNIT=u}\left[,\left\{\begin{matrix}\text{DISPOSE}\\ \text{DISP}\end{matrix}\right\}=\text{p}\right]\text{ [,ERR=s])}$$

u
A logical unit number.

p
A character constant that determines the disposition of the file. Its values are 'SAVE', 'KEEP', 'DELETE', and 'PRINT'.

s
The label of an executable statement.

If you specify either 'SAVE' or 'KEEP', the file is retained after the unit is closed. If you specify 'PRINT', the file is submitted to the line printer spooler and is not deleted. If you specify 'DELETE', the file is deleted. For scratch files, the default is 'DELETE'; for all other files, the default is 'SAVE'. 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 or printed, nor can a file opened for read-only access be printed or deleted.

For example:

```
CLOSE(UNIT=1,DISPOSE='PRINT')
```

This statement closes the file on unit 1 and submits the file for printing.

```
CLOSE(UNIT=J,DISPOSE='DELETE',ERR=99)
```

This statement closes the file on unit J and deletes it.

REWIND

9.3 REWIND STATEMENT

The REWIND statement repositions a currently open sequential file at the beginning of the file. It has the form:

```
REWIND u
```

u
A logical unit number.

The unit number must refer to an open file on disk or magnetic tape. For example:

```
REWIND 3
```

This statement repositions logical unit 3 to the beginning of currently open file.

You must not issue a REWIND statement for a file that is open for direct access.

BACKSPACE

9.4 BACKSPACE STATEMENT

The BACKSPACE statement repositions a currently open sequential file at the beginning of the preceding record. When the next I/O statement for the unit executes, that record is available for processing.

The BACKSPACE statement has the form:

```
BACKSPACE u
```

u
A logical unit number.

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' or 'APPEND' access.

FIND

9.5 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 form:

```
FIND (u'r)
```

u
A logical unit number.

r
The direct access record number.

The unit number must refer to an open 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.

9.6 ENDFILE STATEMENT

The ENDFILE statement writes an end-file record to the specified unit. It has the form:

```
ENDFILE u
```

u
A logical unit number.

An end-file record can be written only to sequentially accessed, sequentially organized files containing variable length records.

For example:

```
ENDFILE 2
```

This statement outputs an end-file record to logical unit 2.

DEFINE FILE

9.7 DEFINE FILE STATEMENT

The DEFINE FILE statement describes direct access files that are associated with a logical unit number. (See Section 9.1 for a preferred way to do this.) 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)] ...
```

u
A logical unit number.

m
A numeric expression that specifies the number of records in the file.

n
A numeric expression that specifies the length, in 16-bit words (2 bytes), of each record.

U
Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv
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.

DEFINE FILE specifies that a file containing m fixed length records of n 16-bit words each exists, or is to exist, on logical unit u. The records in the file are numbered sequentially from 1 through m.

DEFINE FILE must be executed before the first direct access I/O statement that refers to the specified file.

DEFINE FILE also establishes the integer variable asv as the associated variable of the 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. Since the associated variable always points to the next sequential record in the file (unless it is redefined by an assignment, input, or FIND statement), direct access I/O statements can perform sequential processing of the file, 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.

APPENDIX A

CHARACTER SETS

A.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
Δ	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 A-1.

A.2 ASCII CHARACTER SET

Table A-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 A-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

A.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. It 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:

$nRc_1c_2...c_n$

n
: An unsigned, nonzero integer constant that states the number of characters to follow.

c
: 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)
6RΔTOΔΔΔ	

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 B

FORTRAN LANGUAGE SUMMARY

B.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 priority)
	.GE.	Greater than or equal to	
	.LT.	Less than	
	.LE.	Less than or equal to	
	.EQ.	Equal to	
	.NE.	Not equal to	

Data Type	Operator	Operation	Operates upon:
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	
	.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. 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	

B.2 STATEMENTS

The following summarizes the statements available in the VAX-11 FORTRAN IV-PLUS 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 the manual that describes each statement in detail.

Form	Effect	Manual Section
ACCEPT	See READ, Formatted Sequential See READ, List-Directed	7.2.1 7.3.1
Arithmetic/Logical/Character Assignment		3.1, 3.2, 3.3
v=e		
v	A variable name, an array element name, or a character substring name	
e	An expression	
	Assigns the value of the arithmetic, logical, or character expression to the variable.	

Arithmetic Statement Function 6.2.1

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

f — A symbolic name

p — A symbolic name

e — 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.

ASSIGN s TO v 3.4

s — Label of an executable statement

v — An integer variable name

Associates the statement label s with the integer variable v for later use in an assigned GO TO statement.

BACKSPACE u 9.4

u — An integer expression

Backspaces the currently open file on logical unit u one record.

BLOCK DATA [nam] 5.10

nam — A symbolic name

Specifies the subprogram that follows as a BLOCK DATA subprogram.

CALL f[([a][,[a]]...)] 4.5 6.2

f — A subprogram name or entry point

a — 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 (p[,p]...) 9.2

p — One of the following forms:

```
UNIT = e
DISPOSE = 'SAVE'
DISPOSE = 'KEEP'
DISPOSE = 'DELETE'
DISPOSE = 'PRINT'
ERR = s
```

e — An integer expression

s — Label of an executable statement

Closes the specified file. DISPOSE can be abbreviated DISP.

COMMON [/[cb]/] nlist [[,]/[cb]/nlist]... 5.4

cb — A common block name

nlist — 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 4.4

Causes no processing.

DATA nlist/clist/[[,] nlist/clist/]... 5.7

nlist — A list of one or more variable names, array names, array element names, or character substring names separated by commas. Subscript expressions and substring expressions must be constant.

clist — 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[,ERR=s])[list] 7.7

c	An integer expression
f	A format specifier
b	A variable name, array name, array element name, or character substring name
s	A label of an executable statement
list	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.

DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]... 9.7

u	An integer expression
m	An integer expression
n	An integer expression
v	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 words of a single record, U is a fixed argument, and v is the associated variable.

DIMENSION a(d)[,a(d)]... 5.3

a(d)	An array declarator

Specifies storage space requirements for arrays.

DO s [,] v = el,e2[,e3] 4.3

s The label of an executable statement

v A variable name

ei,e2,e3 Numeric expressions

Executes the DO loop by performing the following steps:

1. Set v = el
2. Execute all statements through statement number s
3. Evaluate v = v+e3
4. Repeat steps 2 through 3 for the following iterations:

MAX (1, INT((e2 - el)/e3) + 1)

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.

ELSE IF (e) THEN 4.2.3

e 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[,ERR=s]) [list] 7.7

c An integer expression

f A format specifier

b A variable name, array name, array element name, or substring name

s A label of an executable statement

list 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 — 4.9

Delimits a program unit.

ENDFILE u — 9.6

u — An integer variable or constant

Writes an end-file record on logical unit u.

END IF — 4.2.3

Terminates block IF construct. See IF THEN.

END=s,ERR=s — 7.1.6, 9.1.7

s — A label of an executable statement

Transfers control on end-of-file or error condition. This is an optional element in each type of I/O statement and allows the program to transfer to statement number s when an end-of-file (END=) or error (ERR=) condition occurs.

ENTRY nam [([p[,p]...])] — 6.2.4

nam — A subprogram name

p — A symbolic name

Defines an alternative entry point within a subroutine or function subprogram.

EQUIVALENCE (nlist)[,(nlist)]... — 5.5

nlist — 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 constants.

Assigns each of the names in nlist the same storage location.

EXTERNAL [*]v[,[*]v]... 5.6

v A subprogram name

Defines the names specified as user-defined subprograms.

FIND (u'r) 9.5

u An integer expression

r An integer expression

Positions the file on logical unit u to record r and sets the associated variable to record number r.

FORMAT (field specification,...) 8.1 - 8.7

Describes the format in which one or more records are to be transmitted; a statement label must be present.

[typ] FUNCTION nam[*n][([p[,p]...])] 6.2.2

typ A data type specifier

nam A symbolic name

*n A data type length specifier

p A symbol name

Begins a function subprogram, indicating the program name and any dummy argument names (p). An optional type specification can be included.

GO TO s 4.1.1

s A label of an executable statement

Transfers control to statement number s.

GO TO (slist)[,] e 4.1.2

slist — A list of one or more statement labels separated by commas

e — 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, etc.). If e is less than 1 or greater than the number of statement labels present, no transfer takes place.

GO TO v [[,](slist)] 4.1.3

v — An integer variable name

slist — 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.

IF (e) s1,s2,s3 4.2.1

e — An expression

s — A label of an executable statement

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 — An expression

st — Any executable statement except a DO 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 Logical expressions

block 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.

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 A data type specifier

a Either a single letter, or two letters in alphabetical order separated by a hyphen (i.e., 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' 1.5

'file specification'
A character constant

Includes the source statements in the compilation from the file specified.

OPEN(par[,par]...) 9.1

par — A keyword specification in one of the following forms:

```
key
key = value
```

key — A keyword, as described below.

value — Depends on the keyword, as described below.

Keyword	Values
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'
ASSOCIATEVARIABLE	v
BLOCKSIZE	e
BUFFERCOUNT	e
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'
{DISPOSE} {DISP }	'SAVE' or 'KEEP' 'PRINT' 'DELETE'
ERR	s
EXTENDSIZE	e
FORM	'FORMATTED' 'UNFORMATTED'
INITIALSIZE	e
MAXREC	e
NAME	c
NOSPANBLOCKS	-
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'
READONLY	-
RECORDSIZE	e
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'
SHARED	-
TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'
UNIT	e
USEROPEN	p

c — A character expression, numeric array name, numeric variable name, numeric array element name, or Hollerith constant

e — A numeric expression

p — A program unit name

s — A statement label

v — An integer variable name

Opens a file on the specified logical unit according to the parameters specified by the keywords.

PARAMETER p=c [,p=c]... 5.8

p — A symbolic name

c — A constant or compile-time constant expression

Defines a symbolic name for a constant.

PAUSE [disp] 4.7

disp — 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, Formatted Sequential 7.2.2
See WRITE, List-Directed 7.3.2

PROGRAM nam 5.9

nam — A symbolic name.

Specifies a name for the main program.

READ (u,f[,END=s][,ERR=s])[list] 7.2.1

READ f[,list]

ACCEPT f[,list]

u — An integer expression

f — A format specifier

s — A label of an executable statement

list — 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 (u'r,f[,ERR=s])[list] 7.5.1

u	An integer expression
r	An integer expression
f	A format specifier
s	A label of an executable statement
list	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(u[,END=s][,ERR=s])[list] 7.4.1

u	An integer expression
s	A label of an executable statement
list	An I/O list

Reads one unformatted record from unit u, and assigns values to the elements in the list.

READ(u'r[,ERR=s])[list] 7.6.1

u	An integer expression
r	An integer expression
s	A label of an executable statement
list	An I/O list

Reads record r from unit u, and assigns values to the elements in the list.

READ (u,*[,END=s][,ERR=s])list 7.3.1

READ *,list

ACCEPT *,list

u	An integer expression
*	Denotes list-directed formatting
s	A label of an executable statement
list	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.

RETURN [i] 4.6

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 u 9.3

u	An integer expression

Repositions logical unit u to the beginning of the currently opened file.

STOP [disp] 4.8

disp	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	A symbolic name
p	A symbolic name

Begins a subroutine subprogram, indicating the program name and any dummy argument names (p).

TYPE See WRITE, Formatted Sequential 7.2.2
See WRITE, List-Directed 7.3.2

Type Declaration 5.2

typ v[,v]...

typ — 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
DOUBLE PRECISION
COMPLEX
COMPLEX*8
CHARACTER*len
CHARACTER*(*)

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 *(*).

The symbolic names (v) are assigned the specified data type.

VIRTUAL a(d)[,a(d)]... 5.3

Equivalent to the DIMENSION statement.

WRITE (u,f[,ERR=s])[list] 7.2.2

PRINT f[,list]

TYPE f[,list]

u — An integer expression

f — A format specifier

s — A label of an executable statement

list — 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 (u'r,f[,ERR=s])[list] 7.5.2

u	An integer expression
r	An integer expression
f	A format specifier
s	A label of an executable statement
list	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 (u[,ERR=s])[list] 7.4.2

u	An integer expression
s	A label of an executable statement label
list	An I/O list

Writes one unformatted record to unit u containing the values of the elements in the list.

WRITE (u'r[,ERR=s]) [list] 7.6.2

u	An integer expression
r	An integer expression
s	A label of an executable statement label
list	An I/O list

Writes record r to unit u containing the values of the elements in the list.

```
WRITE(u,*[,ERR=s])list                                    7.3.2

PRINT *,list

TYPE *,list
```

u	An integer expression
*	Denotes list-directed formatting
s	A label of an executable statement
list	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.

B.3 LIBRARY FUNCTIONS

Table B-1 lists the VAX-11 FORTRAN IV-PLUS generic functions and processor-defined functions (listed in the column headed "PDF Name"). Superscripts in the table refer to notes, which follow the table.

Table B-1 Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
Square Root[1] $a^{1/2}$	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Natural Logarithm[2] $\log_e a$	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm[2] $\log_{10} a$	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Exponential e^a	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Sine[3] Sin a	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Cosine[3] Cos a	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent[3] Tan a	1	TAN	TAN DTAN	Real Double	Real Double
Arc Sine[4,5] Arc Sin a	1	ASIN	ASIN DASIN	Real Double	Real Double
Arc Cosine[4,5] Arc Cos a	1	ACOS	ACOS DACOS	Real Double	Real Double
Arc Tangent[5] Arc Tan a	1	ATAN	ATAN DATAN	Real Double	Real Double

Table B-1 (Cont.) Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
Arc Tangent[5,6] Arc Tan a_1/a_2	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine Sinh a	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine Cosh a	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent Tanh a	1	TANH	TANH DTANH	Real Double	Real Double
Absolute value[7] |a|	1	ABS	ABS DABS CABS IIABS JIABS	Real Double Complex Integer*2 Integer*4	Real Double Real Integer*2 Integer*4
		IABS	IIABS JIABS	Integer*2 Integer*4	Integer*2 Integer*4
Truncation[8] [a]	1	INT	IINT JINT IIDINT JIDINT	Real Real Double Double	Integer*2 Integer*4 Integer*2 Integer*4
		IDINT	IIDINT JIDINT	Double Double	Integer*2 Integer*4
		AINT	AINT DINT	Real Double	Real Double
Nearest Integer[8] [a+.5*sign(a)]	1	NINT	ININT JNINT IIDNNT JIDNNT	Real Real Double Double	Integer*2 Integer*4 Integer*2 Integer*4
		IDNINT	IIDNNT JIDNNT	Double Double	Integer*2 Integer*4
		ANINT	ANINT DNINT	Real Double	Real Double

Table B-1 (Cont.) Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
Fix[9] (real-to-integer conversion)	1	IFIX	IIFIX JIFIX	Real Real	Integer*2 Integer*4
Float[9] (integer-to-real conversion)	1	FLOAT	FLOATI FLOATJ	Integer*2 Integer*4	Real Real
Double Precision Float[9] (integer-to-double conversion)	1	DFLOAT	DFLOTI DFLOTJ	Integer*2 Integer*4	Double Double
Conversion to[9] Single Precision	1	SNGL	– SNGL FLOATI FLOATJ	Real Double Integer*2 Integer*4	Real Real Real Real
Conversion to[9] Double Precision	1	DBLE	DBLE – DFLOTI DFLOTJ	Real Double Integer*2 Integer*4	Double Double Double Double
Real Part of Complex	1	–	REAL	Complex	Real
Imaginary Part of Complex	1	–	AIMAG	Complex	Real
Complex From Two Reals	2	–	CMPLX	Real	Complex
Complex Conjugate (if a=(X,Y) CONJG (a)=(X,- Y)	1	–	CONJG	Complex	Complex
Double product of Reals $a_1 * a_2$	2	–	DPROD	Real	Double

Table B-1 (Cont.) Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
Maximum $\max(a_1, a_2, \ldots a_n)$ (returns the maximum value from among the argument list; there must be at least two arguments)	n	MAX	AMAX1 DMAX1 IMAX0 JMAX0	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
		MAX0	IMAX0 JMAX0	Integer*2 Integer*4	Integer*2 Integer*4
		MAX1	IMAX1 JMAX1	Real Real	Integer*2 Integer*4
		AMAX0	AIMAX0 AJMAX0	Integer*2 Integer*4	Real Real
Minimum $\min(a_1, a_2, \ldots a_n)$ (returns the minimum value among the argument list; there must be at least two arguments)	n	MIN	AMIN1 DMIN1 IMIN0 JMIN0	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
		MIN0	IMIN0 JMIN0	Integer*2 Integer*4	Integer*2 Integer*4
		MIN1	IMIN1 JMIN1	Real Real	Integer*2 Integer*4
		AMIN0	AIMIN0 AJMIN0	Integer*2 Integer*4	Real Real
Positive Difference $a_1 - (\min(a_1, a_2))$ (returns the first argument minus the minimum of the two arguments)	2	DIM	DIM DDIM IIDIM JIDIM	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
		IDIM	IIDIM JIDIM	Integer*2 Integer*4	Integer*2 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	AMOD DMOD IMOD JMOD	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
Transfer of Sign $\lvert a_1 \rvert$*Sign a_2	2	SIGN	SIGN DSIGN IISIGN JISIGN	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
		ISIGN	IISIGN JISIGN	Integer*2 Integer*4	Integer*2 Integer*4

Table B-1 (Cont.) Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
Bitwise AND (performs a logical AND on corresponding bits)	2	IAND	IIAND JIAND	Integer*2 Integer*4	Integer*2 Integer*4
Bitwise OR (performs an inclusive OR on corresponding bits)	2	IOR	IIOR JIOR	Integer*2 Integer*4	Integer*2 Integer*4
Bitwise Exclusive OR (performs an exclusive OR on corresponding bits)	2	IEOR	IIEOR JIEOR	Integer*2 Integer*4	Integer*2 Integer*4
Bitwise Complement (complements each bit)	1	NOT	INOT JNOT	Integer*2 Integer*4	Integer*2 Integer*4
Bitwise Shift (a_1 logically shifted left a_2 bits)	2	ISHFT	IISHFT JISHFT	Integer*2 Integer*4	Integer*2 Integer*4
Random number[10] (returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1)	1	—	RAN	Integer*4	Real
Length[11] (returns length of the character expression)	1	—	LEN	Character	Integer*4
Index[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 B-1 (Cont.) Generic and Processor-Defined Functions

Functions	Number of Arguments	Generic Name	PDF Name	Type of Argument	Type of Result
ASCII Value[11] (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

Notes for Table B-1

[1] The argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT 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, ALOG10, and DLOG10 must be greater than zero. The argument of CLOG must not be (0., 0.).

[3] The argument of SIN, DSIN, COS, DCOS, DCOS, TAN, and DTAN must be in radians. The argument is treated modulo 2*pi.

[4] The absolute value of the argument of ASIN, DASIN, ACOS, and DACOS must be less than or equal to 1.

[5] The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

[6] The result of ATAN2 and DATAN2 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 SNGL with a real argument and the function DBLE with a double precision argument return the value of the argument without conversion.

[10] The argument for this function must be an integer variable or integer array element. The argument should initially be set to 0. The RAN function stores a value in the argument that it later uses to calculate the next random number. Resetting the argument to 0 regenerates the sequence. Alternate starting values generate different random number sequences.

[11] See Cnapter 6 for additional information on character functions.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

VAX-11 FORTRAN IV-PLUS
Language Reference Manual
AA-D034A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Please cut along this line.

— — Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

— Do Not Tear - Fold Here —