SPERRY UNIVAC

VORTEX FORTRAN IV
Programmer Reference

Mini-Computer Operations

2722 Michelson Drive
P.O. Box C-19504
Irvine, California 92713

98A 9952 042

SPERRY UNIVAC®

VORTEX FORTRAN IV
PROGRAMMER REFERENCE MANUAL

98A 9952 042

FEBRUARY 1978

 Address comments regarding this document to Sperry Univac, Mini-Computer Operations, Publications Department, 2722 Michelson Drive, P.O. Box C-19504, Irvine, California, 92713.

Sperry Univac is a division of Sperry Rand Corporation

Printed in U.S.A.

CHANGE RECORD

Page Number	Issue Date	Change Description
all	12/76	Minor revisions have been incorporated throughout this manual.
various	2/78	Deleted references to Varian.

Change Procedure:

When changes occur to this manual, updated pages are issued to replace the obsolete pages. On each updated page, a vertical line is drawn in the margin to flag each change and a letter is added to the page number. When the manual is revised and completely reprinted, the vertical line and page-number letter are removed.

LIST OF EFFECTIVE PAGES

Page Number	Change in Effect	Page Number	Change in Effect
i thru viii	revised		
1-1 thru 2-8	minor revisions		
2-9	Section 2.4.5 Array Element ordering has been incorporated.		
3-1 thru 9-6	minor revisions		
A-1	addition of ANSI exceptions		
A11	complete revision		

TABLE OF CONTENTS

SECTION 1
INTRODUCTION

SECTION 2
BASIC ELEMENTS

SECTION 3
SPECIFICATION STATEMENTS

CONTENTS

SECTION 4
EXPRESSIONS

SECTION 5
ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENTS

SECTION 6
CONTROL STATEMENTS

SECTION 7
INPUT/OUTPUT STATEMENTS

SECTION 7
INPUT/OUTPUT STATEMENTS *(continued)*

SECTION 8
PROGRAMS AND SUBPROGRAMS

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1
INTRODUCTION

SPERRY UNIVAC Level-G FORTRAN IV is a programming system for the V70/620 series computers and is comprised of a language, library of subprograms, compiler, and a run-time package (program). Programs written in the SPERRY UNIVAC Level-G FORTRAN IV language can be compiled and run under the VORTEX operating systems.

The FORTRAN IV language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. In fact, the name of the language FORTRAN is derived from its primary use: FORmula TRANslating. Source programs written in the FORTRAN language consist of a set of statements constructed from the elements described in this publication. The FORTRAN compiler analyzes the source program statements and transforms them into an object program that is suitable for execution. In addition, when the FORTRAN compiler detects errors in the source program, appropriate error messages are produced.

The principal features of SPERRY UNIVAC Level-G FORTRAN IV include:

- Full compatibility with American National Standards Institute (ANSI) FORTRAN x3.9, 1966, with the exceptions noted in appendix A.
- Alternate Return mechanism.
- Up to seven dimensions.
- Apostrophized string literals as constants, arguments, and format phrases.
- Optional size in Type Specifications.
- ENTRY statement.
- END, ERR I/O specifiers.
- T, Z format phrases.
- Generalized subscripts.
- IMPLICIT statement
- Direct-Access statements
- Mixed-Mode expressions

- Entended display in STOP and PAUSE.
- Initialization of variables in Type Specifications.
- TITLE statement.
- ENCODE/DECODE.
- Hexadecimal constants.

1.1 FORMAT

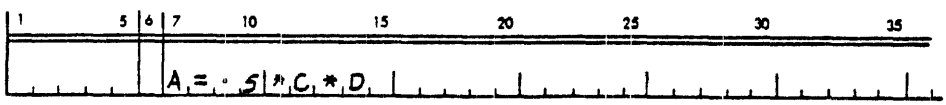
A FORTRAN program consists of a series of statements divided into physical sections called lines that must be coded to a precise grammatical format. FORTRAN statements fall into two broad classes, executable and nonexecutable. Executable statements specify program action; nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arragnement. The statements of a FORTRAN source program are normally written on a standard FORTRAN coding form.

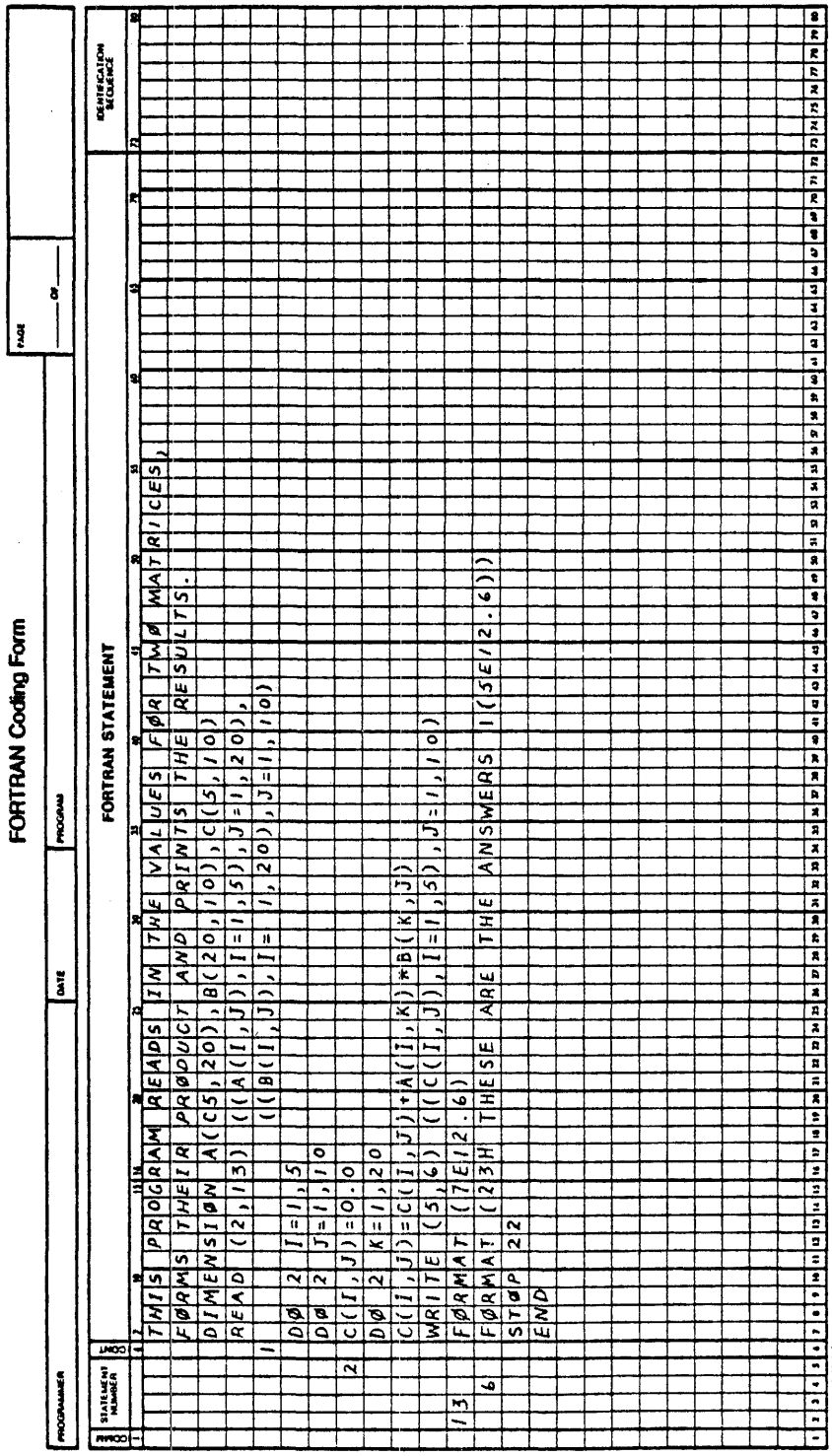
Figure 1-1 is a sample FORTRAN coding form. The coding form includes 80 columns of information. Columns 73 through 80 are reserved for sequencing information, and have no effect upon the generated object program. Columns 1 through 72 contain program information in the format described below.

1.1.1 Initial Line

The first line of each statement is called an initial line. A statement line consists of three fields: statement label field, continuation flag, and statement field. A statement can include an initial line and continuation lines. Statements can have any number of continuation lines as required subject to the following restrictions: DO statements must have the first comma contained on an initial line; and the equals character (=) of a replacement statement or a statement function definition must appear on the initial line. An initial line can contain a statement label in columns 1 through 5. In this case, columnn 6 must contain a zero digit, blank, or space character, and columns 7 through 72 may contain all or part of a statement except for the restrictions noted.

Example





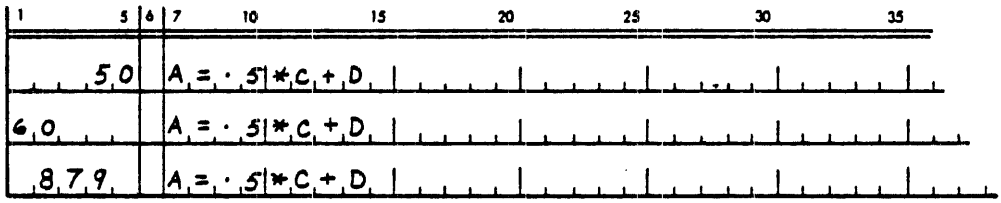
VT12-472

Figure 1-1. Sample FORTRAN Coding Form

1.1.2 Statement Label

The statement label permit statements to be referenced by other portions of a program. A statement label is an integer value in the range 1 to 99999 (leading zeros or blanks are not significant). The initial line of each statement may be given a unique label in columns 1 through 5. The same label cannot be given to more than one statement in a program unit.

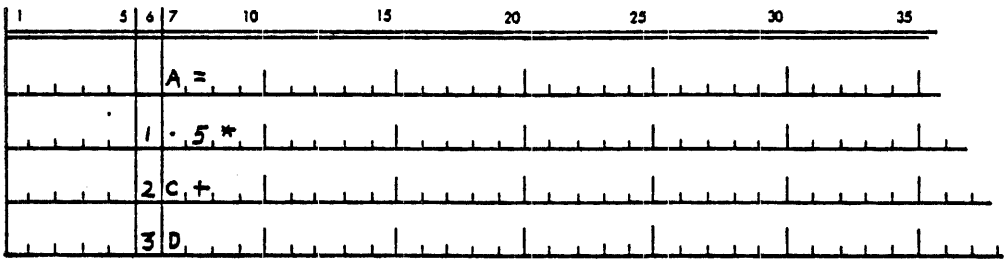
Example



1.1.3 Continuation Line

Continuation lines are used when additional lines of coding are required to complete a statement originating on an initial line. There can be any number of continuation lines per statement. In a continuation line, columns 1 through 5 are blank. Column 6 contains any character other than zero, blank, or space. The continuation of the statement is in columns 7 through 72.

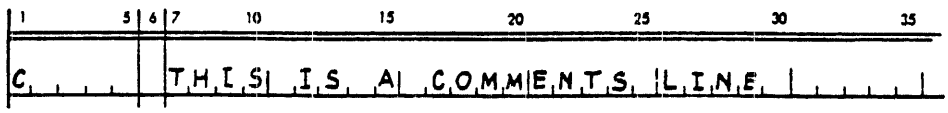
Example



1.1.4 Comment Line

Any line with the character C or an asterisk (*) in column 1 is identified as a comment line. Comments can appear anywhere in a program. All comment lines are ignored by a FORTRAN compiler, except for display purposes. Comments are in columns 2 through 72.

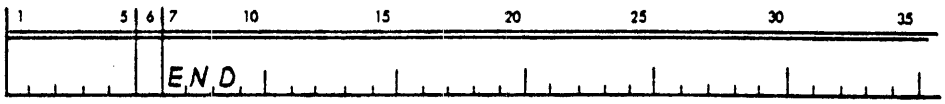
Example



1.1.5 End Line

Any line containing the character blank in columns 1 through 6 and having only the character string END in columns 7 through 72, preceded by, interspersed with, or followed by blank characters, is recognized by the processor as an end line to inform the processor that it has reached the physical end of the program.

Example



1.2 LANGUAGE CONVENTIONS

The FORTRAN language constructs employed in this manual present the general format of the construct with the optional portions of the construct represented by lower-case letters and defined following the specification of the format. The remaining characters in the specification are required portions of the construct being defined. For example:

```
EX(s)MP,LE
```

where

s is a string of from one to three characters, each of which is the character A.

This example consists of the presentation of the general format of a fictitious language construct called an "Example Item." The portion of the construct over which the programmer has control is denoted by the lower-case letter s and is defined below the format specification. The balance of the characters in the example format (i.e., all upper-case letters, the parentheses, and the comma) are required portions of the construct. As defined, valid examples of this sample construct would be:

```
EX(A)MP,LE
EX(A A)MP,LE          (blank characters are ignored in
EX(   AAA)MP,L E      FORTRAN except in string literals)
```

1.3 CHARACTER SET

A FORTRAN program unit is written using the following letters, digits, and special characters:

Letters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z $

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters:

	blank or space
=	equals
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
'	apostrophe

With the exception of the specific uses indicated in the following sections of this manual, a blank character has no meaning, and can be used freely by the programmer to improve the readability of the FORTRAN program.

The following special characters are classified as arithmetic operators and are significant in the unambiguous statement of arithmetic expressions:

+	addition or positive value
-	subtraction or negative value
*	multiplication
/	division
**	exponentiation

The special characters apostrophe ('), equals (=), open parenthesis ((), close parenthesis ()), comma (,), and decimal point (.), have specific application in the syntactical expression of the FORTRAN statement. The following sections of this manual qualify their use in particular statements and expressions.

In addition to the FORTRAN character-set, the SPERRY UNIVAC 70/620 FORTRAN IV system accepts the following characters in Hollerith fields:

"	quotation mark	\	back slash
↑	uparrow	[	left bracket
!	exclamation	]	right bracket
#	number sign	<	less than
%	percent	>	greater than
&	ampersand	?	question mark
;	semicolon	:	colon

1.4 ORGANIZATION

This manual presents the FORTRAN programmer with information directly connected with the FORTRAN language as implemented on the V70 series computers under the VORTEX operating system.

The general discussion in this document proceeds from basic language elements to general FORTRAN program structures. This manual contains the following information:

- Section 1 is an introduction to SPERRY UNIVAC FORTRAN.
- Section 2 discusses constants, variables, and arrays. Primary units of which the language is constructed.
- Sections 3 through 7 discuss FORTRAN expressions and statements. Computation-directed elements of the language.
- Section 8 describes FORTRAN programs and subprograms.
- Section 9 describes VORTEX operating system procedures.

In addition, a number of reference aids (appendices) are provided at the end of this manual.

1.5 BIBLIOGRAPHY

The following gives the stock numbers of Sperry Univac manuals pertinent to the use of FORTRAN (the x at the end of each document number is the revision number and can be any digit 0 through 9):

Title	Number
VORTEX I Reference Manual	98 A 9952 10x
VORTEX II Reference Manual	98 A 9952 24x
V70 TOTAL Data Base Management System Reference Manual	98 A 9952 41x
V70 Series Architecture Reference Manual	98 A 9906 00x

SECTION 2
BASIC ELEMENTS

Constants and variables are distinguished in FORTRAN to identify the nature and characteristics of the values encountered in program execution. A constant is a quantity whose value is explicitly stated. A variable is a numeric quantity referenced by name, rather than by its explicit appearance in a program statement. During the execution of a program, a variable can assume many different values.

2.1 DATA TYPES

The SPERRY UNIVAC 70/620 FORTRAN IV compiler recognizes the following types of data: integer, real, double-precision, complex, logical, and Hollerith. Integer data are precise representations of integral values. Real and double-precision data are approximations of real numbers. Complex data are approximations of complex numbers. Integer, real, and double-precision data may assume positive, negative, or zero values (zero is considered neither positive nor negative).

Integer data may consist of 1-word or 2-word items. Real data may consist of 2-word or 4-word (identical to double-precision) items.

FORTRAN data (variables, arrays, and array elements) are identified by names made up of letter or digit strings of one to six characters, the first character of which is a letter. (The character $ is processed exactly like a letter, but it is reserved for Sperry Univac system names. To avoid conflict, therefore, it is advisable not to use the $ character in names.) Names so identified are implicitly specified as being of Type integer or real by the first character, although this can be changed by an IMPLICIT statement, or in the case of any specified name(s), by an explicit specification using a Type statement. In the absence of such statements, names beginning with the letters I, J, K, L, M, and N denote integers and other names denote real values.

Examples of implicit integer names are (if no Type or IMPLICIT statements are present):

```
I     I2A   MZXF      N5
```

Examples of implicit real-number names are (if no Type or IMPLICIT statements are present):

```
A     B2    F5M79     AAA
```

2.3 CONSTANTS

Constant data are identified explicitly by giving their actual values. Constants do not change in value during program execution. There are three classes of constants -- those that specify

numbers (numerical constants), those that specify truth values (logical constants), and those that specify character strings (Hollerith constants).

Numerical constants are integer, real, double-precision, or complex numbers; logical constants are .TRUE. or .FALSE.; and Hollerith constants are a string of alphameric and/or special characters.

2.3.1 Numeric Constants

A numeric constant can be written either in decimal form or as a hexadecimal (base 16) string.

2.3.1.1 Hexadecimal Constants

The hexadecimal constant consists of the letter Z followed by a string of hexadecimal (base 16) digits. Hexadecimal constants may be used only as data initialization values in DATA statements. The constant has the general form:

Zn

where

n is a hexadecimal digit string

The set of hexadecimal digit values are as follows:

Character	Decimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

The maximum number of digits allowed in a hexadecimal constant depends on the length specification of the variable being initialized. If the number of digits is greater than the maximum, the left-most digits are truncated. If the number is less than the maximum, the

left-most positions are filled with zeros. The following list shows the maximum number of digits for each length specification (in bytes):

Length Specification	Maximum Number of Digits
8	16
4	8
2	4

Examples

```
Z1C49A2F1  represents the bit string:
00011100010010011010001011110001

ZBADFADE  represents the bit string:
10111010110111111101011011110
```

2.3.1.2 Integer Constants

An integer constant is a string of decimal digits without a decimal point, or a hexadecimal string. It can be preceded by a plus (+) or minus (-) sign.

An integer constant may be positive, zero, or negative. If unsigned and nonzero, it is assumed to be positive. If a zero is specified with or without a preceding sign, the sign will have no effect on the value zero. The magnitude must not be greater than the maximum and it may not contain embedded commas. The constant has the general form:

```
    sn
or
    sZh
```

where

s — is the optional signed character (+ or -)

n — is a decimal character string (maximum magnitude is 32767 for 1-word items, and 1073741823 for 2-word items)

h — is a hexadecimal character string (maximum magnitude is Z7FFF for 1-word items, and Z7FFF7FFF for 2-word items)

Example Valid integer constants

```
0
91
173
-1073741823
Z5A
```

Invalid integer constants

27. (contains a decimal point)
3145903612(exceeds the maximum magnitude)
5,396 (contains an embedded comma)
Z100000005(exceeds the maximum magnitude)

2.3.1.3 Real Constants

A real constant is a hexadecimal string, or a decimal real constant, which is defined as follows:

- A basic real constant is written as an integer part, a decimal point, and a fraction part, in that order. Both the integer part and the fraction part are strings of decimal digit characters; either one of these strings may be empty, but not both.
- A decimal exponent is written as the letter E, followed by an optionally signed decimal integer constant.
- A decimal real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or a decimal integer constant followed by a decimal exponent.
- A real constant may occupy either 2- or 4-words (the 4-word real type is indistinguishable from double precision). The range of both is approximately $10^{\pm 38}$.

The format of a 2-word real constant is:

sm.n

or

sZh

where

s denotes an optional sign character.

h is a hexadecimal character string (maximum magnitude Z7FFF7FFF).

m,n represent strings of decimal digits (+ or -). Either m or n (but not both) may be omitted. An alternative form for a real constant, similar to scientific notation is:

smpnEsd

p is an optional decimal point which may be omitted only if n is omitted.

d is a decimal integer constant ≤ 38.

The following are equivalent real constants:

```
2E3
  2.E3
 +2.E+03
Z447D0000
```

Examples

```
17.          -25.620E-1          0.0
51E1          +.42               -479
-479E-3        .35E02
```

The following are invalid real constants:

-1234	No decimal point or E part; interpreted as an integer literal
6.2E + 99	Exceeds maximum size limit
6.2E-99	Smaller than minimum
Z77	Not real format
E5	Exponent part alone not allowed; taken as a variable name
1.2E3.4	Exponent part must be an integer
3E4E5	More than one exponent part
5,432.1	No commas or other punctuation allowed in real constant

2.3.1.4 Double-Precision Constants

A double-precision exponent is identical to the exponent of a real constant, except the letter D is used instead of the letter E. A double-precision constant is indicated by writing a basic real constant followed by a double-precision exponent, an integer constant followed by a double-precision exponent, or a hexadecimal character string.

A double-precision constant may assume positive, negative, and zero values.

Example

```
-3476.2D-4      28.D0      .578D+3

Z814000 0000 0000
```

2.3.1.5 Complex Constants

A complex constant is formed by an ordered pair of signed or unsigned real 2-word constants separated by a comma and enclosed in parentheses.

The real constants in a complex constant can be positive, zero, or negative (if unsigned, they are assumed to be positive). The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

Examples

Valid Complex Constants

`(-5.0E + 03,.16E + 02)`	has the value -5000. + 16.0i
`(4.0E + 03,.16E + 02)`	has the value 4000. + 16.0i
`(4.0E + 03,.16E + 02)`	has the value 4000. + 16.0i
`(2.1,0.0)`	has the value 2.1 + 0.0i
`(Z40C00000,Z40C00000)`	has the value 1.0 + 1.0i

where i equals the square root of -1.

Invalid Complex Constants

`(292704,1.697)`	the real part does not contain a decimal point
`(1.2E113.279.3)`	the real part contains an invalid decimal exponent
`(.003D4,.005D6)`	double-precision constants are invalid

2.3.2 Non-Numeric Constants

There are two kinds of non-numeric constants: logical and Hollerith.

2.3.2.1 Logical Constants

A logical constant specifies the logical value of a variable. There are two logical values: .TRUE. and .FALSE.. Each must be preceded and followed by a period as shown. The logical constants .TRUE. and .FALSE. specify that the value of the logical variable with which they are

associated is true or false, respectively. Logical constants may also be written as a hexadecimal string, for example:

```
.TRUE.: ZFFFF
.FALSE.: Z0
```

2.3.2.2 Hollerith Constants

Hollerith constants are non-empty strings of alphanumeric and/or special characters. If apostrophes delimit the string, a single apostrophe within the string is represented by two apostrophes. If wH precedes the literal, a single apostrophe within the string is represented by a single apostrophe. Blanks within the character string will be considered part of the string.

Hollerith constants can be used in actual argument lists of a subprogram, as data initialization values, or in FORMAT statements. A string enclosed in apostrophes, may also be used in a PAUSE or STOP statement. The constant has the general form:

```
wHs or 's'
```

where

w	is a positive non-zero constant denoting the width of the character string.
s	denotes the character string.

Example

```
24H INPUT/OUTPUT AREA NO.2
'DATA'
'X-COORDINATE Y-COORDINATE Z-COORDINATE'
'3.14'
'DON''T'
5HDON'T
```

2.4 VARIABLES

A FORTRAN variable is a data item, identified by a symbolic name, that occupies a storage area. This section explains the use of variable names, simple variables, arrays, and subscripts.

2.4.1 Variable Names

A FORTRAN variable name is an identifier that consists of a string of one to six alphanumeric characters with the leading character being a letter (including $). Embedded blanks are permitted within variable names but will be removed by the system. A FORTRAN variable name may refer to either a simply variable, or an array.

Variables are classified into the following five fundamental types: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL.

2.4.2 Simple Variables

A simple variable is a single item which is only referenced by the variable name, without subscripting.

2.4.3 Arrays

An array is an ordered set such that each member or element can be referenced by a subscripted array name.

The proper format of an array element reference is:

v (s)

where

v is a variable name and s is a subscript list which is a sequence of arithmetic expressions separated by commas.

A variable name is an array name only if it appears in an appropriate specification statement, such as a DIMENSION, Type or COMMON statement as a declarator. The declarator is used to set the number and maximum size of the dimensions allowed. In a program, an identifier can be used as a simple variable name or an array name, but not both.

Whenever an array name appears in FORTRAN program, the name must be immediately followed by a subscript list, except when it appears in:

*a. A COMMON, DIMENSION, or type statement

b. A DATA statement

c. The list of an I/O statement

d. The dummy argument list of a subprogram.

e. The actual argument list of a subprogram reference

* In this case, an array name may be followed by an array declarator (see section 3.1), which may be of similar format but whose meaning is different.

2.4.4 Subscripts

Each element of an array may be referenced by means of appropriate subscripts. Each entry in a subscript list is evaluated to obtain an integer value. Normally, the minimum value that the subscript of array element can have is one. The maximum is the value specified in the array declarator.

A subscript is an arithmetic expression, or a sequence of arithmetic expressions separated by commas, that is associated with an array name to identify a particular element of the array. The number of subscript expressions in any subscript must be the same as the number of dimensions of the array with whose name the subscript is associated. A subscript is enclosed in parentheses and is written immediately after the array name. A maximum of seven subscript expressions can appear in a subscript.

2.4.5 Array Element Ordering

Elements within an array are ordered by the first subscript, second subscript, etc. For example, suppose the (1-word) integer array IA is a dimensioned as follows:

```
DIMENSION IA(2,3)
```

and is assigned the location 02000. Then the elements of IA would be allocated as follows:

Location	Element
02000	IA(1,1)
02001	IA(2,1)
02002	IA(1,2)
02003	IA(2,2)
02004	IA(1,3)
02005	IA(2,3)

SECTION 3
SPECIFICATION STATEMENTS

Every FORTRAN program or subprogram consists of a sequence of statements terminating with an END line. These statements may be classified into executable and non-executable statements.

An executable statement causes an action at that point in the program when the program is executed.

A non-executable statement supplies information to the compiler when it is processing the FORTRAN statements. In general, these statements specify variable types, initial values, storage allocation, and allow subprograms to be used as actual arguments.

Specification statements are non-executable statements that organize and classify data that will be referred to by other statements in the FORTRAN program. Specification statements include:

DIMENSION	Names and declares the size of an array.
COMMON	Assigns variable and/or named arrays to common storage area.
EQUIVALENCE	Assigns variables and names array to shared storage areas.
EXTERNAL	Declares a name to be external to the program.
IMPLICIT	Specifies the type and length (standard or optional) of all variables, arrays, and user-supplied function whose names begin with a particular letter or range of letters.
Type	Specifies the type and length (standard or optional) of a variable, array, or user-supplied function of a particular name.

Specification statements must precede all other statements except TITLE, BLOCK DATA, FUNCTION, SUBROUTINE, and NAME.

3.1 ARRAY DECLARATORS

Array declarators indicate the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or Type statement. An array declarator has the following format:

```
a(s1,s2,...,sn)
```

where

s — is either on integer constant (si ≥), or an integer variable. The integer variable si can only appear in a subprogram where both si and the corresponding array (a) are formal parameters.

si specifies the maximum value of the ith subscript. The type of each array element is the type associated with the name (a).

n — specifies the number of dimensions in the array (a).

3.2 DIMENSION STATEMENT

A *DIMENSION* statement specifies that the declarator names listed are arrays in the program unit. The DIMENSION statement has the general form

DIMENSION a1,a2,...,an

where

ai — is an array declarator

Example (assuming no Type or IMPLICIT statements are present)

```
DIMENSION A(5),I1(3,6),C(5,10),BIG(10,10,10)
```

Explanation

This specification statement indicates that `A` is a real vector with five elements; `I1` is an integer matrix of size 3 X 6 = 18 elements; `C` is a real matrix of size 5 X 10 = 50 elements; and `BIG` is a real matrix of size 10 X 10 X 10 = 1000 elements.

More than one *DIMENSION* statement can appear in a program.

Note: An array element is referred to by the array name qualified by a subscript to identify the desired element. If the value of this subscript is out of the range specified by the array declarator, the derived computational results will be unpredictable.

3.3 COMMON STATEMENT

The COMMON statement is used to cause the sharing of storage by two or more program units, and to specify the names of variables and arrays that are to occupy this area. Storage sharing can be used for various purposes, e.g., to conserve storage, by avoiding more than one allocation of storage for variables and arrays used by several program units; or to implicitly transfer arguments between a calling program and a subprogram. Arguments passed in a common area are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list.

A given common block name may appear more than once in a COMMON statement, or in more than one COMMON statement in a program unit. All entries within such blocks are strung together in order of their appearance. The length of a common area can be increased by using an EQUIVALENCE statement as long as elements are not added before the established beginning of the COMMON block. The COMMON statement has the general form

```
COMMON /r1/a1,a2,...
```

where

a — is a variable name or array declarator which contains no names that are formal parameters.

r — represents an optional common block name consisting of one through six alphameric characters, the first of which is alphabetic. These names must always be enclosed in slashes.

The form // (with no characters except possibly blanks between the slashes) may be used to denote blank common. If r denotes blank common, the two slashes immediately following the word COMMON, are optional.

Variables or arrays that appear in a calling program or subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the *COMMON* statements. For example, if one program contains the statement:

```
COMMON TABLE
```

as its first *COMMON* statement, and a second program contains the statement:

```
COMMON TREE
```

as its first *COMMON* statement and the two programs are loaded together, the variable names TABLE and TREE refer to the same storage location.

If the main program contains the statement:

```
COMMON A,B,C
```

and a subprogram contains the statement:

```
COMMON X,Y,Z
```

then A shares the same storage location as X, B shares the same storage location as Y, and C shares the same storage location as Z.

Common entries appearing in *COMMON* statements are cumulative in the given order throughout the program; that is, they are cumulative in the sequence in which they appear in all COMMON statements. For example, consider the following two *COMMON* statements:

```
COMMON A,B,C
COMMON G,H
```

These two statements have the same effect as the single statement:

```
COMMON A,B,C,G,H
```

Redundant entries are not allowed. For example, the following statement is invalid:

```
COMMON A,B,C,A
```

Consider the following examples:

Example 1

Calling Program

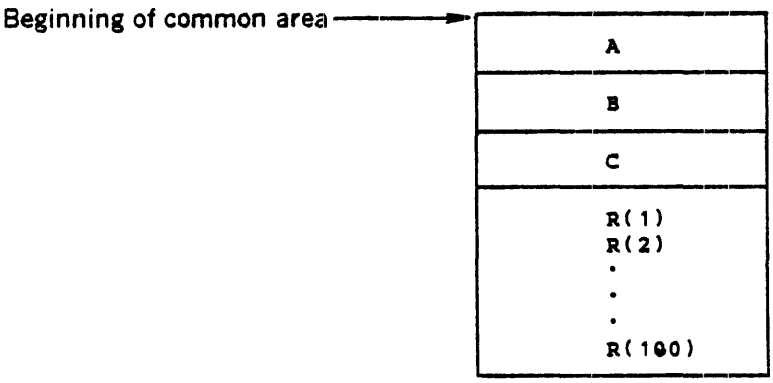
```
COMMON A,B,C,R(100)
REAL A,B,C
INTEGER R
        .
        .
        .
CALL MAPMY
        .
        .
        .
```

Subprogram

```
SUBROUTINE MAPMY
COMMON X,Y,Z,S(100)
REAL X,Y,Z
INTEGER S
        .
        .
        .
```

Explanation

The statement COMMON A,B,C,R(100) in the calling program would cause 106 words to be reserved in the following order:



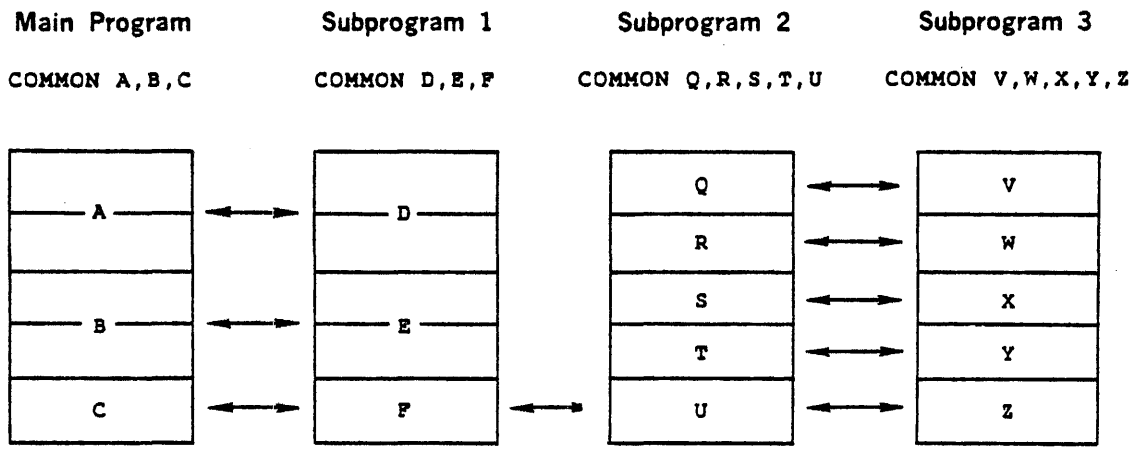
The statement COMMON X, Y, Z, S(100) in the subprogram would then cause the variables X, Y, Z, and S(1),...,S(100) to share the same storage space as A, B, C, and R(1),...,R(100), respectively. Note that values for X, Y, Z, and S(1),...,S(100), because they occupy the same storage locations as A, B, C, and R(1),...,R(100), do not have to be transmitted in the argument list of a CALL statement.

Example 2

Assume a common area is defined in a main program and in three subprograms as follows:

Main Program: COMMON A,B,C (A and B are 4-words, C is 2-words)
Subprogram 1: COMMON D,E,F (D and E are 4-words, F is 2-words)
Subprogram 2: COMMON Q,R,S,T,U (2-words each)
Subprogram 3: COMMON V,W,X,Y,Z (2-words each)

The correspondence of these variables within common can be illustrated as follows:



The main program can transmit values for A, B, and C to subprogram 1, provided that A is of the same type as D, B is of the same type as E, and C is of the same type as F. However, the main program and subprogram 1 cannot, by assigning values to the variables A and B, or D and E, respectively, transmit values to the variables Q, R, S, and T in subprogram 2, or V, W, X, and Y in subprogram 3, because the lengths of their common variables differ. Likewise, subprograms 2 and 3 cannot transmit values to variables A and B, or D and E.

Values can be transmitted between variables C, F, U, and Z, assuming that each is of the same type. With the same assumption, values can be transmitted between A and D, and B and E, and between Q and V, R and W, S and X, and T and Y. Note, however, that assignment of values to A or D destroys any values assigned to Q, R, V, and W, (and vice versa) and that assignment to B or E destroys the values of S, T, X, and Y (and vice versa).

3.3.1 Blank and Labeled COMMON

In the preceding example, the common storage area (common block) established is called a blank common area. That is, no name was explicitly given to that area of storage (the name COMMON is assigned internally to the blank common block and will appear on maps). The variables that appeared in the *COMMON* statements were assigned locations relative to the beginning of the blank common area. However, variables and arrays may be placed in

separate common areas. Each of these separate areas (or blocks) is given a name consisting of one through six alphanumeric characters (the first of which is alphabetic); those blocks which have the same name occupy the same storage space.

Those variables that are to be placed in labeled (or named) common are preceded by a common block name enclosed in slashes. For example, the variables A, B, and C will be placed in the labeled common area, HOLD, by the following statement:

```
COMMON/HOLD/A,B,C
```

In a COMMON statement, blank common can be distinguished from labeled common by preceding the variables in blank common by two consecutive slashes or, if the variables appear at the beginning of the common statement, by omitting any block name. For example, in the following statement:

```
COMMON A,B,C/ITEMS/X,Y,Z//D,E,F
```

the variables A, B, C, D, E, and F will be placed in blank common in that order; the variables X, Y, and Z will be placed in the COMMON area labeled ITEMS.

Blank and labeled common entries appearing in *COMMON* statements are cumulative throughout the program. For example, consider the following two *COMMON* statements:

```
COMMON A,B,C/R/D,E/S/F
COMMON G,H/S/I,J/R/P//W
```

These two statements have the same effect as the single statement:

```
COMMON A,B,C,G,H,W/R/D,E,P/S/F,I,J
```

COMMON is allocated from low to high memory addresses within a common block.

3.3.2 Blank COMMON in VORTEX Foreground/Background

Blank common can be used like labeled common or for communications among foreground tasks. The extent of blank common for foreground tasks is determined at system generation time. The size of the foreground blank common can vary within each task without disturbing the positional relationship of entries, but cannot exceed the limits set at system generation time.

A blank common block for a background task is allocated within the load module. The size of the background blank common can vary within each task, but the combined areas of the load module and common cannot exceed available memory.

Each blank common is accessible only by the corresponding tasks (i.e., foreground tasks use only foreground blank common, and background tasks use only background blank common).

Note: All definitions of labeled and blank common areas for a given load module must be in the first object module loaded.

3.4 EQUIVALENCE STATEMENT

All the elements within a single set of parentheses share the same storage locations. The EQUIVALENCE statement provides an option for controlling the allocation of data storage within a single program unit (see section 2.4.5 for allocation of elements in multidimensional arrays). In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or different types. Equivalence between variables implies storage sharing. Mathematical equivalence of variables or array elements is implied only when they are of the same type, when they share exactly the same storage, and when the value assigned to the storage is of that type. The EQUIVALENCE statement has the general form

EQUIVALENCE (a1,a2,a3,...),...

where

a — Each a is a variable or array element and may not be a dummy argument. The subscripts of array elements may have either of two forms:

If the array element has a single subscript quantity, it refers to the linear position of the element in the array (i.e., its position relative to the first element in the array: 3rd element, 17th element, 259th element).

If the array element is multi-subscripted (with the number of subscript quantities equal to the number of dimensions of the array), it refers to position in the same manner as in an arithmetic statement (i.e., its position relative to the first element of each dimension of the array). In either case, the subscripts themselves must be integer constants.

Since arrays are stored in a predetermined order, equivalencing two elements of two different arrays may implicitly equivalence other elements of the two arrays. The EQUIVALENCE statement must not contradict itself or any previously established equivalences.

Note that the EQUIVALENCE statement is the only statement in which a single subscript may be used to refer to an element (or elements) in a multi-dimensional array.

Two variables in different blocks (common or program) cannot be made equivalent. If a variable that is equivalenced to a variable in the common block is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of the common block (see Example 3, below). The size of the common block cannot be extended so that elements are added before the beginning of the established common block.

Example 1

Assume that in the initial part of a program, an array C of size 100 x 100 is needed; in the final stages of the program C is no longer used, but arrays A and B of sizes 50 x 50 and 100, respectively, are used. The elements of all three arrays are of the type DOUBLE PRECISION. Storage space can then be saved by using the statements:

```
DIMENSION C(100,100),A(50,50),B(100)
EQUIVALENCE (C(1),A(1)),(C(2501),B(1))
```

Explanation

The array A, which has 2500 elements, can occupy the same storage as the first 2500 elements of array C since the arrays are not both needed at the same time. Similarly, the array B can be made to share storage with elements 2501 to 2600 of array C.

Example 2

```
DIMENSION B(5),C(10,10),D(5,10,15)
EQUIVALENCE (A,B(1),C(5,3)),(D(5,10,2),E)
```

Explanation

This EQUIVALENCE statement specifies that the variables A, B(1), and C(5,3) are assigned the same storage locations and that variables D(5,10,2) and E are assigned the same storage locations. It also implies that the array elements B(2) and C(6,3), etc., are assigned the same storage locations. Note that further equivalence specification of B(2) with any element of array C other than C(6,3) is invalid.

Example 3

```
COMMON A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(1))
```

Explanation

This would cause a common area to be established containing the variables A, B, and C. The EQUIVALENCE statement would then cause the variable D(1) to share the same storage location as B, D(2) to share the same storage location as C, and D(3) would extend the size of the common area, in the following manner:

```
A                (lowest location of the common area)
B, D(1)
C, D(2)
   D(3)          (highest location of the common area)
```

The following EQUIVALENCE statement is invalid:

```
COMMON A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(3))
```

because it would force D(1) to precede A, as follows:

```
   D(1)
A, D(2)          (lowest location of the common area)
B, D(3)
C                (highest location of the common area)
```

3.5 VARIABLE TYPING

Initially, the type of all variable and array names are specified implicitly, according to the first character of the name, as follows:

First Character	Type
I - N	Integer (1-word)
All others	Real (2-words)

Those implicit type specifications can be changed by the IMPLICIT statement.

3.5.1 Variable Size Specification

IMPLICIT and Type statements allow a variable size specifier. This is an integer constant which specifies the number of 8-bit bytes allocated to items. Since the V70/620 is a 16-bit word addressable computer, this specifier must be divided by two to get the corresponding word count. The following table lists the permissible options:

Type	Size Specifier	Word/Item
INTEGER	2 (default)	1
	4	2
REAL	4 (default)	2
	8	4
COMPLEX	8	4
LOGICAL	2	1
DOUBLE-PRECISION	8	4

Note: A REAL item with a size specifier of 8 is indistinguishable from a DOUBLE-PRECISION item.

3.5.2 IMPLICIT Statement

The IMPLICIT specification statement enables the user to declare the type of the variable appearing in his program (i.e., integer, real, double-precision, complex, or logical) by specifying that variables beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows the programmer to declare the number of words to be allocated for each item in the group of specified variables.

The IMPLICIT statement has the general form

```
IMPLICIT type *s(a1,a2),...
```

where

type	is one of the following: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.
*s	is optional; and represents a variable size specifier.
a	is a single alphabetic character or a range of characters drawn from the set A, B,...,Z, $, in that order. The range is denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

If the standard length specification (for its associated type) is desired, the *s may be omitted in the IMPLICIT statement. That is, the variables will assume the standard length specification. If the optional length specification is desired, then the *s must be included within the IMPLICIT statement.

Example 1

```
IMPLICIT REAL (A-H, O-$), INTEGER (I-N)
```

Explanation

All variables beginning with the characters I through N are declared as INTEGER. Since no length specification was explicitly given (i.e., the *s was omitted), 1-word (the standard length for INTEGER) is allocated for each variable. All other variables (those beginning with the characters A through H, O through Z, and $) are declared as REAL with 2-words allocated for each.

Example 2

```
IMPLICIT INTEGER*4(A-H), REAL*8(I-K), LOGICAL(L,M,N)
```

Explanation

All variables beginning with the characters A through H are declared as INTEGER with 2-words allocated for each. All variables beginning with the characters I through K are declared as REAL with 4-words allocated for each. Variables beginning with the characters L, M, and N are declared as LOGICAL with 1-word for each.

Since the remaining letters of the alphabet, namely, O through Z and $, are not specified by the IMPLICIT statement, the predefined convention will take effect. Thus, variables beginning with the characters O through Z and $ are declared as REAL, each with a standard length of 2-words.

3.5.3 Explicit Type Statements

The type specification statements explicitly declare the type (INTEGER, REAL, DOUBLE-PRECISION, COMPLEX, or LOGICAL) of a particular variable or array by its name, rather than by its initial character. This differs from the other ways of specifying the type of a variable or array (i.e., predefined convention and the IMPLICIT statement). In addition, the information necessary to allocate storage for arrays (dimension information) may be included within the statement. The statement has the general form

```
type*s a1/x1/,a2/x2/,...
```

where

type	is INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or COMPLEX.
*s	is optional; and, represents a variable size specifier.
a	is a variable, array, array declarator, or function name.
/x/	is optional; and, represents initial data values.

Initial data values may be assigned to variables or arrays by use of /x/ where x is a constant or list of constants separated by commas. The x provides initialization only for the immediately preceding variable or array. The data must be of the same type as the variable or array, except that Hollerith or hexadecimal data may also be used. Lists of constants are used only to assign initial values to array elements. Successive occurrences of the same constant can be represented by the form i* constant, as in the DATA statement. If initial data values are assigned to an array in an Type specification statement, the dimension information for the array must be in the Type statement or in a preceding DIMENSION or COMMON statement. An initial data value may not be assigned to a function name. But, a function name may appear in an explicit Type specification statement. Dummy arguments may not be assigned initial values.

Initial data values cannot be assigned to variables or arrays in blank common. Assigning initial value to variables and arrays in labeled common can only be done within a BLOCK DATA subprogram.

In the same manner in which the IMPLICIT statement overrides the predefined convention, the Type statement overrides the IMPLICIT statement and predefined convention. If the length specification is omitted (i.e., *s), the standard length of the specified type is assumed.

3.5.4 External Statement

When an actual parameter list of a function reference or a subroutine-call contains a function or subroutine name, that name must appear in an EXTERNAL statement in the program in which the reference or call appears.

The form of the EXTERNAL statement is

```
EXTERNAL s1,s2,...,sn
```

where

si is a function or subroutine name

The following are examples of valid EXTERNAL statements

```
EXTERNAL SUB1, SINF
EXTERNAL FRAIL
```

SECTION 4
EXPRESSIONS

Expressions specify the procedure by which a data value is obtained. An expression is any valid constant, variable, function reference, or a combination of these separated by appropriate operators and parentheses, which conforms to the rules given in this section.

Expressions can be divided into three types: arithmetic, logical, and relational. If the value which can represent the result is .TRUE. or .FALSE., then the expression is logical. A relational expression appears only in the context of a logical expression. An expression which yields a numeric quantity is an arithmetic expression.

The operators that can be used by a FORTRAN expression are listed in the table below with a relative precedence assigned to each operator by the compiler (the lowest number has the highest precedence). Since the unary + operator performs no functions, it is not included.

OPERATOR	RELATIVE PRECEDENCE	FUNCTION
**	1	exponentiation
unary-	2	change of sign
/	3	division
*	3	multiplication
-	4	subtraction
+	4	addition
.NE.	5	not equal to
.GE.	5	greater than or equal to
.GT.	5	greater than
.EQ.	5	equal to
.LE.	5	less than or equal to
.LT.	5	less than
.NOT.	6	logical negation
.AND.	7	logical conjunction
.OR.	8	logical disjunction

The occurrence of these operators indicates that an arithmetic, logical, or relational action is to be performed.

4.1 ARITHMETIC EXPRESSIONS

The arithmetic elements are described by the following statements:

PRIMARY — An *ARITHMETIC EXPRESSION* enclosed in parentheses, a constant, a variable reference, an array element reference, or function reference.

FACTOR — A *FACTOR* is a *PRIMARY* or a construct of the form: *PRIMARY***PRIMARY*

EXPRESSIONS

TERM	A *TERM* is a FACTOR or one of the forms: *TERM/FACTOR* *TERM*TERM*
SIGNED TERM	A *TERM* immediately preceded by a + or - sign.
SIMPLE ARITHMETIC EXPRESSION	A *TERM* or two *SIMPLE ARITHMETIC EXPRESSIONS* separated by a + or - sign.
ARITHMETIC EXPRESSION	A *SIMPLE ARITHMETIC EXPRESSION* or a signed *TERM* or either of the preceding immediately followed by a + or - sign and a *SIMPLE ARITHMETIC EXPRESSION*.

A part of an expression is evaluated only if it is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. The range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence. Use of an array element name requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript. An element whose value is not mathematically defined cannot be evaluated.

The following rules represent the derivation of all permissible expressions:

A variable, constant, or function standing alone is an expression.

```
A(1)
JOBNO
217
17.26
SQRT(A+B)
```

If E is an expression whose first character is not an operator, then +E and -E are expressions.

```
-A(1)
+JOBNO
-217
+17.26
-SQRT(A+B)
```

If E is an expression, then (E) is an expression meaning the quantity E taken as a unit.

```
(-A)
-(+JOBNO)
-(X+Y)
(A-SQRT(A+B))
```

If E is an expression whose first character is not an operator, and F is an expression, then: F + E,F-E,F*E,F/E and F**E are all expressions.

```
-(B(I,J)+SQRT(A+B(K,L)))
-(B(I+E,3*J+K)+A)
1.7E-2**(X+5.0)
```

4.1.1 Arithmetic Operators

More complicated arithmetic expressions containing two or more primaries may be formed by using arithmetic operators that express the computation(s) to be performed.

The arithmetic operators are as follows:

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

All desired computations must be specified explicitly. That is, if more than one primary appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two variables A and B will not be multiplied if written:

```
AB
```

In fact, AB is regarded as a single variable with a two-letter name.

If multiplication is desired, the expression must be written as follows:

```
A*B or B*A
```

No two arithmetic operators may appear consecutively in the same expression. For example, the following expressions are invalid:

```
A*/B and A*-B
```

The expression A*-B could be written correctly as

```
A*(-B)
```

In effect, -B will be evaluated first and then A will be multiplied with it.

4.1.2 Order of Computation

Computation is performed according to the hierarchy of operations shown in the following list.

Operation	Hierarchy
Evaluation of functions	1st
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th

This hierarchy is used to determine which of two sequential operations is performed first. If the first operator is higher than or equal to the second, the first operation is performed. If not, the second operator is compared to the third, etc.

For example, in the expression A*B+C*D**I, the operations are performed in the following order:

1.	`A*B`	Call the result X (multiplication)	`(X+C*D**I)`
2.	`D**I`	Call the result Y (exponentiation)	`(X+C*Y)`
3.	`C*Y`	Call the result Z (multiplication)	`(X+Z)`
4.	`X+Z`	Final operation (addition)	

If there are sequential exponentiation operators, the evaluation is from right to left. Thus, the expression:

```
A**B**C
```

is evaluated as follows:

1.	`B**C`	Call the result Z
2.	`A**Z`	Final operation

A unary plus or minus has the same hierarchy as a plus or minus in addition or subtraction. Thus,

`A=-B` is treated as `A=0-B`

`A=-B*C` is treated as `A=-(B*C)`

`A=-B+C` is treated as `A=(-B)+C`

4.1.3 Use of Parentheses

Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used. This is equivalent to the definition above, since a parenthesized expression is a primary.

For example, the following expression:

```
B/((A-B)*C)+A**2
```

is effectively evaluated in the following order:

1.	`A-B`	Call the result W	`B/(W*C)+A**2`
2.	`W*C`	Call the result X	`B/X+A**2`
3.	`B/X`	Call the result Y	`Y+A**2`
4.	`A**2`	Call the result Z	`Y+Z`
5.	`Y+Z`	Final operation	

4.1.4 Type and Length of Results of Expressions

The type and length of the result of an operation depends upon the type and length of the two operands (primaries) involved in the operation. The below matrix shows the type and length of the result of the operations +, -, *, and /.

TYPE and LENGTH MATRIX*

First Operand \ Second Operand	INTEGER (1)	INTEGER (2)	REAL (2)	REAL (4)	COMPLEX (4)
INTEGER (1)	INTEGER (1)	INTEGER (2)	REAL (2)	REAL (4)	COMPLEX (4)
INTEGER (2)	INTEGER (2)	INTEGER (2)	REAL (2)	REAL (4)	COMPLEX (4)
REAL (2)	REAL (2)	REAL (2)	REAL (2)	REAL (4)	COMPLEX (4)
REAL (4)	REAL (4)	REAL (4)	REAL (4)	REAL (4)	REAL (4)
COMPLEX (4)	COMPLEX (4)	COMPLEX (4)	COMPLEX (4)	COMPLEX (4)	COMPLEX (4)

* in this matrix, all numeric entries present element size in words

A *PRIMARY* of any type may be exponentiated by an *INTEGER PRIMARY* and the resulting factor is of the same type as that of the element being exponentiated. A *REAL* or *DOUBLE-PRECISION PRIMARY* may be exponentiated by a *REAL* or *DOUBLE-PRECISION PRIMARY*. The resultant *FACTOR* is of type *REAL* if both *PRIMARIES* are *REAL*, and otherwise of type *DOUBLE PRECISION*. These are the only cases for which use of the exponentiation operator is defined. Valid combinations for exponentiation are:

Base	Exponent
REAL	*REAL, INTEGER* or *DOUBLE PRECISION*
INTEGER	*INTEGER*
DOUBLE PRECISION	*REAL, INTEGER* or *DOUBLE PRECISION*
COMPLEX	*INTEGER*

4.2 LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, logical subscripted variable, or logical function reference, the value of which is always a truth value (i.e., either .TRUE. or .FALSE.).

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of the three following forms:

a. Relational operators combined with arithmetic expressions whose type is *INTEGER, REAL,* or *DOUBLE PRECISION.*

b. Logical operators combined with logical constants (.TRUE. and .FALSE.), logical variables, subscripted logical variables, or logical function references.

c. Logical operators combined with either or both forms of the logical expressions described in items a and b.

Item a is discussed in the following section, Relational Operators; items b and c are discussed in the section entitled Logical Operators.

4.2.1 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator and has the value .TRUE. or .FALSE. as the relation is true or false.

The six relational operators, each of which must be preceded and followed by a period, are as follows:

Relational Operator	Definition
.GT.	Greater than ($>$)
.GE.	Greater than or equal to ($\geq$)
.LT.	Less than ($<$)
.LE.	Less than or equal to ($\leq$)
.EQ.	Equal to ($=$)
.NE.	Note equal to ($\neq$)

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions whose type is *INTEGER, REAL,* or *DOUBLE PRECISION* can be combined by relational operators. For example, assuming the type of variable has been specified as follows:

Variable Names	Type
ROOT, E, Q	*REAL* variables
A, I, F	*INTEGER* variables
L	*LOGICAL* variable
C	*COMPLEX* variable

then, the following illustrates valid and invalid logical expressions using the relational operators.

Example

Valid Logical Expressions Using Relational Operators:

```
(ROOT*Q).GT.E

A.LT.I

E**2.7.EQ.(5.*ROOT+4.)

57.9.LE.(4.7+E)

.5.GE.9.*ROOT

E.EQ.27.3E+05
```

Invalid Logical Expressions Using Relational Operators:

`C.LT.ROOT`	Complex quantities can never appear in logical expressions.
`C.GE.(2.7,5.9E3)`	Complex quantities can never appear in logical expressions.
`E**2.EQ97.1E9`	Missing period immediately after the relational operator.
`.GT.9`	Missing arithmetic expression before the relational operator.

4.2.2 Logical Operators

The three logical operators, each of which must be preceded and followed by a period, are as follows: (A and B represent logical expressions).

Logical Operator	Definition
`.NOT.`	.NOT.A - if A is .TRUE., then .NOT.A has the value .FALSE.; if A is .FALSE., then .NOT.A has the value .TRUE.

Logical Operator	Definition
.AND.	A.AND.B - if A and B are both .TRUE., then A.AND.B has the value .TRUE.; if either A or B or both are .FALSE., then A.AND.B has the value .FALSE.
.OR.	A.OR.B - if either A or B - or both are .TRUE., then A.OR.B has the value .TRUE.; if both A and B are .FALSE., then A.OR.B has the value .FALSE.

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the type of variable has been specified as follows:

Variable Names	Type
ROOT, E, Q	*REAL* variables
A, I, F	*INTEGER* variables
L, W	*LOGICAL* variables
C	*COMPLEX* variable

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Examples

Valid Logical Expressions:

```
(ROOT*Q.GT.E).AND.W

L.AND..NOT.(I.GT.F)

(E + 5.9E2.GT.2.*E).OR.L

.NOT.W.AND..NOT.L

L.AND..NOT.W.OR.I.GT.F

(E**F.GT.ROOT).AND..NOT.(I.EQ.A)
```

Invalid Logical Expressions:

`E.AND.L`	E is not a logical expression.
`.OR.W`	.OR. must be preceded by a logical expression.
`NOT.(A.GT.F)`	missing period before the logical operator .NOT.
`(C.EQ.I).AND.L`	a complex variable may never appear in a logical expression.
`L.AND..OR.W`	the logical operators .AND. and .OR. must always be separated by a logical expression.
`.AND.L`	.AND. must be preceded by a logical expression.

4.2.3 Order of Computations

Where parentheses are omitted, or where the entire logical expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of Functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.	5th
.NOT.	6th
.AND.	7th
.OR.	8th

For example, the expression:

```
(A.GT.D**B.AND..NOT.L.OR.N)
```

is effectively evaluated in the following order.

1.	`D**B`	Call the result W (exponentiation)
2.	`A.GT.W`	Call the result X (relational operator)
3.	`.NOT.L`	Call the result Y (highest logical operator)
4.	`X.AND.Y`	Call the result Z (second highest logical operator)
5.	`Z.OR.N`	Final operation

4.2.4 Use of Parentheses

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the most

deeply nested parentheses (that is, the innermost pair of parentheses) is effectively evaluated first. For example, the logical expression:

```
((I.GT.(B+C)).AND.L)
```

is effectively evaluated in the following order.

1.	`B+C`	Call the result X
2.	`I.GT.X`	Call the result Y
3.	`Y.AND.L`	Final operation

The logical expression to which the logical operator .NOT. applies must be enclosed in parentheses if it contains two or more quantities. For example, assume that the values of the logical variables, A and B, are .FALSE. and .TRUE., respectively. Then the following two expressions are not equivalent:

```
.NOT.(A.OR.B)
.NOT.A.OR.B
```

In the first expression, A.OR.B is evaluated first. The result is .TRUE., but .NOT. (.TRUE.) implies .FALSE.. Therefore, the value of the first expression is .FALSE..

In the second expression, .NOT.A is evaluated first. The result is .TRUE.; but .TRUE.OR.B implies .TRUE.. Therefore, the value of the second expression is .TRUE..

SECTION 5
ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENTS

Each arithmetic or logical statement defines a numerical or a logical calculation. These FORTRAN statements closely resemble a conventional algebraic equation; however, the equal sign specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign. These statements have the general form

```
a = b
```

where

a is a variable or array element.

b is an arithmetic or logical expression.

If b is a logical expression, a must be a logical variable or array element. If b is an arithmetic expression, a must be a integer, real, double-precision, or complex variable or array element. Table 5-1 gives the conversion rules used for placing the evaluated result of arithmetic expression b into variable a.

Assume that the type of the following data items has been specified as:

Symbolic Name	Type	Length Specification
I, J, W	Integer variables	2,2,1
A, B	Real variables	2,2,4,4
C, D	Double-precision variables	4,4
E	Complex variable	4
F(1),...,F(5)	Real array elements	2
G, H	Logical variables	2,2

Then the following examples illustrate valid arithmetic statements using constants, variables, and array elements of different types:

Statements	Description
`A = B`	The value of A is replaced by the current value of B.
`W = B`	The value of B is truncated to a 1-word integer value, and this value replaces the value of W.
`A = I`	The value of 2-word integer (I) is converted to a real value, and this result replaces the value of A.

`I = I + 1`	The value of `I` is repalced by the value of `I + 1`.
`A = C*D`	The most significant part of the product of C and D replaces the value of A.

Examples of logical assignment statements are:

Statement	Description
`G = .TRUE.`	The value of `G` is replaced by the logical constant `.TRUE.`.
`H = .NOT.G`	If `G` is `.TRUE.`, the value of `H` is replaced by the logical constant `.FALSE.`. If `G` is `.FALSE.`, the value of `H` is replaced by the logical constant `.TRUE.`.
`G = 3..GT.I`	The value of `I` is converted to a real value; if the real constant `3.` is greater than this result, the logical constant .TRUE. replaces value of `G`. If `3.` is not greater than `I`, the logical constant `.FALSE.` replaces the value of `G`.

Table 5-1. Conversion Rules for the Arithmetic Assignment Statement a = b

<table>
<tr><th colspan="2" rowspan="2">type of a \ type of b</th><th colspan="2">INTEGER</th><th rowspan="2">REAL</th><th rowspan="2">DOUBLE PRECISION</th><th rowspan="2">COMPLEX</th></tr>
<tr><th>1-word</th><th>2-word</th></tr>
<tr><td rowspan="2">INTEGER</td><td>1-word</td><td>Assign</td><td>Assign Low-order word</td><td colspan="2" rowspan="2">Fix and assign</td><td rowspan="2">Fix and assign real part; imaginary part not used.</td></tr>
<tr><td>2-word</td><td>Extended Sign To high order word</td><td>Assign</td></tr>
<tr><td colspan="2">REAL</td><td colspan="2">Float and assign</td><td>Assign</td><td>DP evaluate and Real assign</td><td>Assign real part; imaginary part not used.</td></tr>
<tr><td colspan="2">DOUBLE PRECISION</td><td colspan="2">DP float and assign</td><td>DP evaluand assign</td><td>Assign</td><td>DP evaluate and assign real part; imaginary not used.</td></tr>
<tr><td colspan="2">COMPLEX</td><td colspan="2">Float and assign to real part; imaginary set to zero</td><td>Assign to real part; imaginary part set to zero</td><td>DP evaluate and assign real part; imaginary part set to</td><td>Assign</td></tr>
</table>

Notes:

1. **Assign** means transmit the resulting value, without change.

2. **Real Assign** means transmit to a as much precision of the most significant part of the resulting value as a REAL datum can contain.

3. **Fix** means truncate the fractional portion of the resulting value and transform it to the form of an integer.

4. **Float** means transform the resulting value to the form of a REAL datum retaining in the process as much precision of the value as a REAL number can contain.

5. **DP Float** means transform the resulting value to the form of a DOUBLE-PRECISION datum.

6. An expression of the form E = (A,B) where E is a complex variable and A and B are real variables, is invalid. The mathematical function subprogram CMPLX can be used for this purpose.

7. **DP Evaluate** means evaluate the expression, then DP Float.

8. **Assign Low-Order Word** means to perform the equivalent of loading the 2-word integer **b** in registers AB, performing a LASL 15, and storing the A-register in **a**.

9. **Extend Sign To High-Order Word** means to perform the equivalent of loading 1-word integer **b** into the A register, clearing bit 15 of the B-register, performing a LASR 15, and storing registers AB in **a**.

SECTION 6
CONTROL STATEMENTS

Normally, FORTRAN statements are executed sequentially. That is, after one statement has been executed, the statement immediately following it is executed. This section discusses certain statements that may be used to alter and control the normal sequence of execution of statements in the program.

The control statements are: *GO TO, IF, CALL, RETURN, CONTINUE, PAUSE, STOP,* and *DO.*

6.1 GO TO STATEMENTS

GO TO statements permit transfer of control to an executable statement by label. Control may be transferred either unconditionally or conditionally. The GO TO statements are:

a. Unconditional GO TO statement

b. Computed GO TO statement

c. Assigned GO TO statement

6.1.1 Unconditional GO TO Statement

This GO TO statement causes control to be transferred to a specified statement. Every subsequent execution of this GO TO statement results in a transfer to that same statement. Any executable statement immediately following this statement should have a statement label; otherwise it can never be referred to or executed. This statement has the general form

```
GO TO xxxxx
```

where

xxxxx	is the label of the executable statement in the same program unit to which control will be transferred.

Example

```
      GO TO 72
        .
   71 V7 = HQ(5)+Y**L
        .
        .
        .
   72 V7 = HQ(4)+X**J
```

Explanation

In this example, execution of the GO TO 72 statement causes subsequent statements to be bypassed. Execution is resumed with statement labeled 72.

6.1.2 Computed GO TO Statement

This statement causes control to be transferred to the statement labeled x1, x2, x3,...,xn, depending on whether the current value of i is 1, 2, 3, ..., or n, respectively. If the value of i is outside the range $1 \leq i \leq n$, the result is underlined. This statement has the general form

```
GO TO (x1,x2,x3,...,xn), i
```

where

xi	is the label of an executable statement in the program unit containing the GO TO statement.
i	is an integer variable (not an array element) which must be given a value before the GO TO statement is executed.

The computed GO TO statement causes control to be transferred to the statement labeled x1, x2, x3,...,xn, depending on whether the current value of i is 1, 2, 3, ..., or n, respectively. If the value of i is outside the range $1 \leq i \leq n$, the result is undefined. No run-time check of the contents of i is done. If i is not completely controlled by the program unit containing the GO TO (e.g., if it were to be read in from an external I/O device), explicit range checking of i should be coded into the program.

Example

```
    GO TO (25,10,7,10), ITEM
345 C = 7.02
      .
      .
      .
  7 C = E**2+A
      .
      .
      .
 25 L = C
      .
      .
      .
 10 B = 21.3E02
```

Explanation

In this example, if the value of the integer variable ITEM is 1, statement 25 will be executed next. If ITEM is equal to 2 or 4, statement 10 is executed next, and so on.

6.1.3 ASSIGN and Assigned GO TO

The assigned GO TO statement causes control to be transferred to the statement labeled x1, x2, x3,..., or xn, depending on whether the current assignment of N is x1, x2, x3,..., or xn, respectively. For example, in the statement:

```
GO TO N, (10,25,8)
```

If the current assignment of the integer variable N is statement label 8, then the statement labeled 8 is executed next. If the current assignment of N is statement label 10, the statement labeled 10 is executed next. If N is assigned statement label 25, statement 25 is executed next.

At the time of execution of an assigned GO TO statement, the current value of N must have been defined to be one of the values x1, x2,...,xn by the previous execution of an ASSIGN statement. The value of the integer variable m is not the integer statement label; ASSIGN 10 to I is not the same as I = 10.

Any executable statement immediately following this statement should have a statement label; otherwise it can never be referred to or executed. This statement has the general form

```
ASSIGN i TO m
       .
       .
       .
GO TO m, (x1,x2,x3,...,xn)
```

where

i	is the label of an executable statement. It must be one of the labels x1,x2,x3,...,xn.
x	is the label of an execuatable statement in the program unit containing the GO TO statement.
m	is a 1-word integer variable (not an array element) which is assigned one of the statement labels: x1, x2, x3,...,xn.

Example 1

```
       .
       .
       .
   ASSIGN 50 TO NUMBER
10 GO TO NUMBER, (35,50,25,12,18)
       .
       .
       .
50 A = B + C
       .
       .
       .
```

Explanation

In Example 1, statement 50 is executed immediately after statement 10.

Example 2

```
          .
          .
          .
      ASSIGN 10 TO ITEM
          .
          .
          .
13 GO TO ITEM, (8,12,25,50,10)
          .
          .
          .
 8 A = B + C
          .
          .
          .
10 B = C + D
   ASSIGN 25 TO ITEM
   GO TO 13
          .
          .
          .
25 C = E**2
          .
          .
          .
```

Explanation

In Example 2, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

6.2 IF STATEMENTS

The following IF statements permit a programmer to change the sequence of statement execution, depending on the value of an arithmetic or logical expression.

6.2.1 Arithmetic IF

The arithmetic IF statement causes control to be transferred to the statement labeled x1,x2, or x3 when the value of the arithmetic expression (a) is less than, equal to, or greater than zero, respectively. This statement has the general form

```
IF (a) x1,x2,x3
```

where

a — is an arithmetic expression of any type except complex.

x — is the label of an executable statement in the program unit containing the IF statement.

Any executable statement immediately following this statement should have a statement label; otherwise it can never be referred to or executed.

Example

```
      IF (A(J,K)**3-B)10, 4, 30
40 D = C**2
         .
         .
         .
 4 D = B + C
         .
         .
         .
30 C = D**2
         .
         .
         .
10 E = (F*B)/D+1
         .
         .
         .
```

Explanation

In this example, if the value of the expression (A(J,K)**3-B) is negative, the statement labeled 10 is executed next. If the value of the expression is zero, the statement labeled 4 is executed next. If the value of the expression is positive, the statement labeled 30 is executed next.

6.2.2 Logical IF

The logical IF statement is used to evaluate the logical expression (a) and to execute or skip statement s depending on whether the value of the expression is true or false, respectively. This statement has the general form

IF (a) s

where

a — is any logical expression.

s is any executable statement except a DO statement or another logical IF statement. The statement s may not have a statement label.

Example 1

```
      .
      .
      .
   IF(A.LE.0.0) GO TO 25
   C = D + E
   IF(A.EQ.B) ANSWER = 2.0*A/C
   F = G/H
      .
      .
      .
25 W = X**Z
      .
      .
      .
```

Explanation

In the first statement, if the value of the expression is true (i.e., A is less than or equal to 0.0), the statement GO TO 25 is executed next and control is passed to the statement labeld 25. If the value of the expression is false (i.e., A is greater than 0.0), the statement GO TO 25 is ignored and control is passed to the second statement.

In the third statement, if the value of the expression is true (i.e., A is equal to B), the value of ANSWER is replaced by the value of the expression (2.0*A/C) and then the fourth statement is executed. If the value of the expression is false (i.e., A is not equal to B), the value of ANSWER remains unchanged and the fourth statement is executed next.

Example 2

Assume that P and Q are logical variables.

```
      .
      .
      .
IF(P.OR..NOT.Q)A=B
C = B**2
      .
      .
      .
```

Explanation

In the first statement, if the value of the expression is true, the value of A is replaced by the value of B and the second statement is executed next. If the value of the expression is false, the statement A = B is skipped and the second statement is executed.

6.3 CALL STATEMENT

The execution of the CALL statement causes the specified subroutine to be executed. The CALL statement arguments must agree in number, type, and order of appearance with the dummy arguments in the SUBROUTINE statement except that subroutines used as actual arguments have no type, and Hollerith constants may be associated with dummy arguments of any type. The statement has the general form

```
CALL name (a1,a2,a3,...)
```

where

name	is the name of a SUBROUTINE subprogram.
a	is an actual argument that is being supplied to the SUBROUTINE subprogram. The argument may be a variable, array element, array name, Hollerith constant, or arithmetic or logical expression. Each a may also be of the form &n: where n is a statement label

If the alternate RETURN mechanism is not used, control will be returned to the first executable statement following the CALL statement upon execution of the RETURN statement in the subroutine. Examples of calling sequences to subroutines are shown below.

Example

```
      CALL TEST (A,I)
      CALL EXIT
      CALL NEXT (A, &50)
           .
           .
           .
50    I=2
```

Explanation

The first example will transfer execution control to the subroutine TEST and include the parameters or arguments A and I in the subroutine. The second example will cause execution control to be transferred to the subroutine EXIT. Any arguments required for execution of EXIT are self-contained in the logic of the subroutine or, in a COMMON block.

The third example will transfer execution to the subroutine NEXT, providing access to name A and statement label 50. If NEXT exits via a RETURN statement, control will return to the statement following CALL NEXT. If NEXT exits via a RETURN 1, control will return to the statement labelled 50.

6.4 RETURN STATEMENT

The execution of a RETURN statement results in the exit from a subprogram. The normal sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next statement following the CALL in the calling program. It is also possible to return to any numbered statement in the calling program by using a return of the type RETURN i. The value of i must be within the range of the argument list. Returns of the type RETURN i may only be made in a SUBROUTINE subprogram. Returns of the type RETURN may be made in either a SUBROUTINE or FUNCTION subprogram. The statement has the general form

```
RETURN
  or
RETURNi
```

where

i — is a 1-word integer constant or variable whose value, say n, denotes the n-th asterisk in the argument list of a SUBROUTINE statement.

A RETURN statement may only appear in a procedure subprogram unit and each procedure subprogram must contain at least one RETURN statement.

6.5 PAUSE STATEMENT

The execution of the PAUSE statement causes the unconditional suspension (SUSPND) of the object program being executed pending operator action. To resume the suspended task, input the operator-communication key-in request RESUME. Execution starts with the next statement after the PAUSE statement. The statement has the general form

```
PAUSE op
```

where

op — is optional; and, if present it represents a string of one to five octal digits, or

is a Hollerith constant enclosed in apostrophes.

When executed, the following message is output to the SO device before the task is suspended.

```
task name PAUSE op
```

To resume, enter the OPCOM directive: RESUME, task name

6.6 STOP STATEMENT

The execution of the STOP statement causes the unconditional termination of the execution of the object program being executed. The statement has the general form

STOP op

where

op is optional; and, if present it represents a string of one to five digits, or

is a Hollerith constant enclosed in apostrophes.

When executed, the following message is output to the SO device before the task is terminated.

```
task name STOP op
```

6.7 CONTINUE STATEMENT

CONTINUE is a statement that may be placed anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution. It may be used as the last statement in the range of a DO in order to avoid ending the DO loop with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing any of these forms. This statement has the general form

CONTINUE

Example 1

```
      .
      .
      .
   DO 30 I = 1, 20
7  IF (A(I)-B(I))  5,30,30
5  A(I) =A(I) +1.0
   B(I) = B(I) -2.0
      .
      .
      .
   GO TO 7
30 CONTINUE
   C = A(3) + B(7)
      .
      .
      .
```

Explanation

In Example 1, the CONTINUE statement is used as the last statement in the range of the DO in order to avoid ending the DO loop with the statement GO TO 7.

Example 2

```
      .
      .
      .
   DO 30 I=1,20
   IF(A(I)-B(I))5,40,40
5  A(I) = C(I)
   GO TO 30
40 A(I) = 0.0
30 CONTINUE
      .
      .
      .
```

Explanation

In Example 2, the CONTINUE statement provides a branch point enabling the programmer to bypass the execution of statement 40.

6.8 DO STATEMENT

The DO statement controls repetitive execution of a group of statements. The number of repetitions depends on the value of a control variable. The statements assumes one of the following forms

```
DO n i = m1, m2, m3
        or
DO n i = m1, m2
```

where

n — is the statement label of an executable statement. This statement, called the terminal statement of the associated *DO* must physically follow and be in the same program unit as the *DO* statement. The terminal statement may not be a *GO TO* of any form, arithmetic *IF, RETURN, STOP, PAUSE,* or another *DO* statement, nor a logical *IF* statement containing one of these forms.

i — is an integer variable name, identified as the control variable.

m1 — is an integer constant or variable name, whose value is greater than or equal to 1. m1 is called the initial parameter. At start of execution of the DO loop, the control variable i is loaded with m1.

m2 is an integer constant or variable name, whose value is greater than or equal to m1. m2 is called the terminal parameter. When the terminal statement is executed and the control variable i is incremented by m3, the DO loop will terminate if i is greater than m2.

m3 is an integer constant or variable, whose value is greater than 1. If m3 is not present in the statement, it is implicitly assigned a value of 1. m3 is called the incrementation parameter. When the terminal statement is executed, the control variable is incremented by m3.

Associated with each *DO* statement is a range that is defined to be those executable statements from and including the first executable statement following the *DO*, to and including the terminal statement defined by the *DO*. A special situation, called nesting, occurs when the range of a *DO* contains another *DO* statement. In this case, the range of the contained *DO* must be a subset of the range of the containing *DO*. There is no limit to the nesting of *DO* statements.

Execution of a DO range proceeds as follows:

a. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

b. The range of the *DO* is executed.

c. After the terminal statement is executed, the control variable of the most recently executed *DO* statement associated with the terminal statement is incremented by the value of its associated incrementation parameter.

d. If the value of the control variable is greater than the value represented by its associated terminal parameter, the *DO* is said to be satisfied, and the control variable becomes undefined. If not, return to step 2.

e. If more than one other *DO* statement refers to the same terminal statement, the control variable of the next most recently executed *DO* statement is incremented by the value represented by the associated incrementation parameter and so on until all *DO* statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a *DO* by execution of a *GO TO* statement or an arithmetic *IF* statement, that is other than by satisfying the *DO*, the control variable of the *DO* is defined and is equal to the most recent attained value.

A *GO TO* or arithmetic *IF* statement may not cause control to pass into the range of a *DO* from outside its range. When a procedure reference occurs in the range of a *DO*, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial, terminal, and incrementation parameters of a *DO* may not be redefined during the execution of the range of that *DO*.

If a statement is the terminal statement of more than one *DO* statement, the label of that terminal statement may not be used in any *GO TO* or arithmetic *IF* statement that occurs anywhere but in the range of the most deeply contained *DO* with that terminal statement.

Example

```
DO 607 K1 = 2, ID, 3
```

Explanation

The foregoing statement would cause K1, the control variable, to be set to the value of the initial parameter, 2. Execution would proceed at the statement immediately following, down to and including the statement identified by the label 607. After each execution of the loop, K1 is incremented by the incrementation parameter, 3, and evaluated in relation to the current value of the terminal parameter, ID. If the value of K1 > ID, execution control is transferred to the statement following that identified by the label 607; otherwise, the *DO* cycle is repeated.

Example illustrating *DO* nesting:

```
      WRITE (MX,8)
      L = 0
      DO 150 J = 1,K
      DO 140 I = 1,M
      L = L + 1
140   D(I) = V(L)
150   WRITE (MX,9)J,(D(I),I = 1,M)
      CALL LOAD (M,K,R,V)
C     PRINT FACTOR MATRIX
      WRITE (MX,10)K
      DO 180 I = 1,M
      DO 170 J = 1,K
      L = M*(J-1)+1
170   D(J) = V(L)
180   WRITE (MX,11)I,(D(J),J = 1,K)
      IF (K-1) 185, 185, 188
      185  WRITE (MX,19)K
           GO TO 100
      188  CALL VARMX (M,K,V,NC,TV,B,T,D)
```

SECTION 7
INPUT/OUTPUT STATEMENTS

This section explains the following four types of input/output statements:

a. Sequential READ/WRITE statements

b. Direct access I/O statements

c. Auxiliary Input/Output statements

d. ENCODE and DECODE statements

READ statements provide a program with the means of receiving information from external sources. WRITE statements allow the transmission of program data to external sources. Auxiliary I/O statements set-up, manipulate, and control external sources. These external sources may be devices such as magnetic tape and parer tape handlers, typewriters, and punch card processor. ENCODE and DECODE statements do not refer to any external source, but manipulate memory arrays as if they were READ/WRITE buffers.

There are two modes of READ/WRITE statements: sequential and direct access. Sequential READ/WRITE statements are used for storing and retrieving data sequentially. These statements are device independent and can be used for files on either sequential or direct access devices.

The direct access READ/WRITE statements are used to store and retrieve data in an order specified by the user. These statements can be used only for a file on a direct access storage device.

I/O List: READ/WRITE statements in FORTRAN are primarily concerned with the transfer of data between storage locations defined in a FORTRAN program and records which are external to the program. DECODE and ENCLODE perform operations that are subsets of READ and WRITE, respectively. They transfer data between items in an an I/O list and a buffer, under format control, but no physical I/O data transfer between the buffer and an external device is performed.

On READ, data is transferred from an external device to a memory buffer. On DECODE, the buffer is specified in the statement. In both cases, data is taken from the buffer, possibly transformed under format control, and placed into storage locations that are note necessarily contiguous.

On WRITE or ENCODE, data is gathered from diverse storage locations and placed into a buffer. On WRITE, this buffer is transferred to an external device. An I/O list is used to specify which storage locations are used. The I/O list can contain variable names, array elements, array names, or a form called an implied DO (see below). No function references or arithmetic expressions are permitted in an I/O list, except in subscripts of array elements in the list. If a function reference is used in a subscript. the function may not perform input/output.

If a variable name or array element appears in the I/O list, one item is transmitted between a storage location and a record.

If an array name appears in the list, the entire array is transmitted in the order in which it is stored. (If the array has more than one dimension, it is stored in ascending storage locations, with the value of the first subscript quantity increasing most rapidly and the value of the last increasing least rapidly (see section 2.4.5 for ordering of array elements.)

Implied DO: If an implied DO appears in the I/O list, the variables, array elements, or arrays specified by the implied DO are transmitted. The implied DO specification is enclosed in parentheses. Within the parentheses are one or more variables, array elements, or array names, separated by commas, with a comma following the last name, followed by indexing parameters i = m1, m2, m3. The indexing parameters are as defined for the DO statement. Their range is the list of the DO-implied list and, for input lists, i, m1, m2, and m3 may appear within that range only in subscripts.

For example, assume that A is a variable and that B, C, and D are 1-dimensional arrays each containing 20 elements. Then the statement:

```
WRITE (6) A, B, (C(I), I=1,4), D(4)
```

writes the current value of variable A, the entire array B, the first four elements of the array C, and the fourth element of D. (The 6 following the WRITE is the FORTRAN unit number.) If the subscript (I) were not included with array C, the entire array would be written four times.

Implied DO's can be nested if required. For example, to read an element into array B after values are read into each row of a 10 x 20 array A, the following would be written:

```
READ (5) ((A(I,J), J=1,20), B(I), I=1,10)
```

The order of the names in the list specifies the order in which the data is transferred between the record and the storage locations.

Formatted and Unformatted Records: Data can be transmitted either under control of a FORMAT statement or without the use of a FORMAT statement.

When data is transmitted with format control, the data in the record is coded in a form that can be read by the programmer or which satisfies the needs of machine representation. The transformation for input takes the character codes and constructs a machine representation of an item. The output transformation takes the machine representation of an item and contructs character codes suitable for printing. Most transformations involve numeric representations that require base conversion. To obtain format control, the programmer must include a FORMAT statement in the program and must give the statement number of the FORMAT statement in the READ or WRITE statement specifying the input/output operation.

When data is transmitted without format control, no FORMAT statement is used. In this case, there is a one-to-one correspondence between internal storage locations and external record positions. A typical use of unformatted data is for information that is written out during a program, not examined by the programmer, and then read back in later in the program, or in another program, for additional processing.

For formatted data, the I/O list and the FORMAT statement determine the form of the record.

7.1 FORTRAN UNIT NUMBERS

Except for ENCODE and DECODE, all I/O statements contain an external I/O device specifier, u, which is called a FORTRAN unit number. This specifier (u) is always a 1-word integer in the range of $0 \leq u < 256$.

At execution time, u must become associated with a VORTEX logical unit number or mass storage file. This is discussed in detail in the following sections.

Note: For all VORTEX system concepts which are referenced in the following sections, refer to the appropriate section in the VORTEX I and VORTEX II reference manuals.

7.1.1 Implicitly Opened Files

Implicitly opened files are files whose FORTRAN unit number (u) has not been defined by a FORTRAN CALL V$OPEN, CALL V$OPNB, or DEFINE FILE statement, or by JCP directives. In this case, the FORTRAN unit number (u) is simply used as the logical unit number (u cannot refer to a mass storage device).

Example

```
WRITE(21) A
```

Explanation

Assuming that FORTRAN unit number 21 has not been defined by a DEFINE FILE, CALL V$OPEN, CALL V$OPNB statement, or by JCP directives, the above statement, at execution time, will simply transfer the contents of A to an output buffer and execute a VORTEX WRITE call to output this buffer to logical unit number 21.

7.1.2 JCP-Opened Background Files

JCP-opened background files are mass storage files which are opened by use of the JCP directives /ASSIGN and /PFILE. It is assumed in the following example that the FORTRAN unit numbers referenced have not been defined by a CALL V$OPEN or a CALL V$OPNB statement.

Example

```
/ASSIGN,PI,25
/PFILE,PI,,FILE1
/FORT
     INTEGER PI
     DATA PI/4/
     READ (PI) A
          .
          .
          .
```

Explanation

In the above example, the FORTRAN unit number has been given in the integer variable name PI. The JCP directives have assigned logical unit name PI (whose logical unit number is 4) to logical unit 25, which is assumed to be an RMD partition containing the file name, FILE1; and opended the PI global FCB on the name FILE1. The FORTRAN execution-time routines will perofrm a VORTEX READ on logical unit 4, using the PI global FCB, and transfer the first 2-words of the input buffer into real item A.

Note: This process can only be used by FORTRAN programs executing in background; and,

a. The FORTRAN unit number must correspond to a global FCB.

b. Only unkeyed files may be referenced.

c. The JCP directives must be executed prior to execution of the FORTRAN program.

7.1.3 Files Opened by CALL V$OPEN and V$OPNB

This section explains the use of CALL V$OPEN and V$OPNB, and associated statements in opening sequential and random access files.

7.1.3.1 CALL V$OPEN and V$CLOS Statements

The CALL V$OPEN statement is used for general reference of mass storage files, from either background or foreground. The general form of CALL V$OPEN is

```
CALL V$OPEN(u,l,n,m)
```

where

u is the FORTRAN unit number.

l is the logical unit number.

n is the name of a 13-word array containing the file name in words 8, 9, 10, and the protect key in the low-order byte of word 3.

m is the VORTEX OPEN mode: 0 = OPEN/rewind
1 = OPEN/no rewind

Files opened with the CALL V$OPEN statement are generally sequential, but random access can be achieved by manipulating word 4 of array n. Word 4 specifies the current record position, and is incremented automatically after every READ or WRITE on the corresponding FORTRAN unit number.

Example

```
      INTEGER     FUN
      DIMENSION   NAME(13)
      DATA        FUN,LUN,MODE/10,20,0/
      DATA        NAME(3),NAME(8),NAME(9),NAME(10)/
     1            2H A,2HFI,2HLE,2HX//
      CALL        V$OPEN(FUN,LUN,NAME,MODE)
      WRITE       (FUN,1)
    1 FORMAT      ('RECORD HEADER')
          .
          .
          .
```

Explanation

This example will open the file named FILEXY, which is defined on the mass storage partition with protect key A. Logical unit 20 is assigned to this partition. Since the mode specifies rewind, the file will be positioned at its start, so the text string 'RECORD HEADER' will be written at the start of the first record. If the WRITE statement had been preceded by a NAME(4) = 5 statement, the text string would have been written on the fifth record.

CALL V$CLOS Statement

The CALL V$CLOS statement is associated with CALL V$OPEN, in that it is used to end V$OPEN. The general form of CALL V$CLOS is

```
CALL V$CLOS(u,m)
```

where

u is the FORTRAN unit number defined by a CALL V$OPEN statement.

m is the VORTEX CLOSE mode: 0 = CLOSE/leave
1 = CLOSE/update

7.1.3.2 CALL V$OPNB and V$CLSB Statements

The CALL V$OPNB statement extends the power of the V$OPEN call. It performs blocking on mass storage devices, or variable record I/O on other type devices. The general form of CALL V$OPNB is

```
CALL V$OPNB(u,l,n,m,r,b,f)
```

where

u	is the FOTRAN unit number.
l	is the logical unit number.
n	is the name of a 14-word array containing the file name in words 8, 9, 10, and the protect key in word 3.
m	is the VORTEX OPEN mode: 0 = OPEN/rewind 1 = OPEN/leave
r	is the logical record size in words.
b	is the name of a blocking buffer array.
f	is the read before write flag: 0 = no read before write 1 = read before write

Note: The first four parameters (u, l, n, and m) are identical to those of V$OPEN, except for n, which requires an additional word.

In using FORTRAN support routines, a sufficient size blocking buffer (b) should be provided. CALL V$OPNB associates this blocking buffer with a FORTRAN unit number u; therefore, the contents of b should be not modified, or used as a blocking buffer for another V$OPNB call.

If a CALL V$CLSB is executed on u, the buffer is released and can be used for any desired purpose, including as a blocking buffer for another CALL V$OPNB.

The rules for defining the sizes of b in words are listed below, where:

a, s, Q, R are integers
Q(a/b) = integer quotient of a/s
R = a - bQ(a/s) = integer remainder of a/s

For u not a mass storage file:

s = r

For u a mas storage file:

$r < 120$ and $R(120/r) = 0 : s = 120$
$r < 120$ and $R(120/r) \neq 0 : s = 240$
$r \geq 120$ and $R(r/120) = 0 : s = r$
$r \geq 120$ and $R(r/120) = 1 : s = 120*(Q\ (r/120)+1)$
$r \geq 120$ and $R(r/120) > 1: s = 120*(Q(r/120)+2)$

A mass storage file opened by a CALL V$OPNB statement is processed as a sequence of physically contiguous logical records, each one r words in length. These logical records cross physical record boundaries (without wasted space) except possibly as the file ends. Input and output is buffered through the user-supplied blocking buffer b.

Since VORTEX physical I/O is performed with buffer b, the file must be large enough to do I/O with the last logical record. Thus, when creating a file, allocate space for one additional logical record than will be required.

On a WRITE operation to a mass storage file (where r is not a multiple or factor of 120 words), data on the file can be overwritten unless the read-before-write flag is set. In some situations, however, such as initial file writing in a strictly sequential fashion, read-before-write is unnecessary and slow.

Example (mass storage file):

```
DIMENSION IARR(14),IBUF(360),IX(200),IY(200)
DATA IARR(3),IARR(8),IARR(9),IARR(10)/2H X,2HAB,2HC, 2H  /
CALL V$OPNB(100,10,IARR,0,200,IBUF,1)
IARR(4) = 10
READ(100) IX
IARR(4) = 9
READ(100) IY
```

Explanation

This is an example of a large logical record (200 words) on a mass storage file. It demonstrates random access. The statement IARR(4) = 10 requests that the file be positioned at the tenth logical record. The statement READ(100) IX will read and transfer the 200 words in the tenth logical record from FORTRAN unit 100 to array IX. The last two statements position FORTRAN unit 100 to logical record 9, and read logical record 9 into array IY.

The CALL V$OPNB statement above has associated FORTRAN unit number 100 with the file named ABC, on a partition with protect key X, to which logical unit number 10 is assigned.

Example (mass storage file):

```
DIMESNION IARR(14),IBUF(120),IX(10),IY(10)
DATA IARR(3),IARR(8),IARR(9),IARR(10)/2H X, 2HAB, 2HC  /
CALL V$OPNB(100,10,IARR,0,10,IBUF,0)
```

```
IARR(4) = 10
READ(100) IX
IARR(4) = 9
READ(100) IY
```

Explanation

This is the same example as before, but with a logical record size of 10 words. Note that in this case there would only be one physical I/O: A read of the first 120-word record of file ABC into blocking buffer IBUF. Since logical records 9 and 10 are both in this physical record, the statement READ(100) IY would cause only a 10-word memory to memory transfer from IBUF to IY, with no external I/O.

Example (not mass storage file):

```
DIMENSION IARR(14),IBUF(43),IX(43)
CALL V$OPNB(100,19,IARR,0,43,IBUF,0)
WRITE(100) IX
```

Explanation

This example demonstrates writing a record of arbitrary size (here 43 words) to a non-mass storage device, perhaps a magnetic tape.

CALL V$CLSB Statement

The CALL V$CLSB statement is associated with CALL V$OPNB, in that it is used to end V$OPNB. The general form of CALL V$CLSB is

CALL V$OPNB(u,m)

where

u is the FORTRAN unit number

m is the VORTEX CLOSE MODE: 0 = CLOSE/leave
1 = CLOSE/update

Note: V$CLSB will undefine u and flush the blocking buffer.

7.1.3.3 V$OPEN and V$OPNB Restrictions

The number of files of this type open at any given time is unlimited, except by memory and permissible range of FORTRAN unit numbers, which allow only numbers 0-255. Active FORTRAN unit numbers (those defined by V$OPEN or V$OPNB and not yet undefined by V$CLOSE or V$CLSB) must be unique, and blocking buffers allocated to active FORTRAN unit

numbers (by a CALL V$OPNB) must be totally dedicated to this function. Different FORTRAN unit numbers may be assigned to the same partition, or even to the same file, but this is not advised for files opened with a CALL V$OPNB, for some of the file data may be in a blocking buffer, and not accessible to another program.

These files will accept the Auxiliary I/O statements, ENDFILE, BACKSPACE, and REWIND. ENDFILE will do a CLOSE/update and reopen.

The VORTEX end-of-file indicator must be used with caution on mass storage files opened with a CALL V$OPNB, for it indicates physical record end-of-file, which may have no relation to the logical record structure. It is best for the user to devise his own indicator on such files.

7.1.4 Direct Access Files

These are files opened by a DEFINE FILE statement, which is discussed in a later section.

7.2 EXTERNAL DEVICES

VORTEX FORTRAN is generally independent of the characteristics of the external device associated with a given FORTRAN unit number. Physical record size and console devices are two important exceptions.

7.2.1 Physical Record Size

FORTRAN associates a physical record size with each FORTRAN unit number (u). For files opened with a CALL V$OPNB statement, it is the blocking buffer size (as shown in section 7.1). For files opened with a CALL V$OPEN statement, it is 120-words, except when u is assigned to the same partition as SI, in which case it is 40-words.

For Direct-Access I/O, it is the record size specified in the DEFINE FILE statement.

For implicitly opened files, FORTRAN associates a physical record size to each device, according to its name, as follows:

First two Characters	Physical Record Size (in words)
CP	40
CR	40
CT	40
CP	66
TY	40
All others	60

This number is used to breakup an I/O request into separate VORTEX physical I/O requests. This is normal for an unformatted I/O request, which may input or output a large array,

VORTEX FORTRAN will also do this for formatted I/O requests, although the user should explicitly control records by using '/' and ')' format specifiers.

7.2.2 Console Devices

Console devices are handled in a special way, in that record input is automatically padded out with blanks, if a partial record is input.

Devices whose names begin with CT or TY are flagged as console devices. On a READ, the operator can always terminate input with a carriage return, and the record will be padded out to its end with blanks.

Example

```
      DIMENSION IA(80)
      READ(4,1) IA
    1 FORMAT(80A1)
```

Explanation

If this section of program is executed, and FORTRAN unit number 4 is a console device, and the operator enters AB cr, FORTRAN will transfer the characters `A` and `B` to `IA(1)` and FWFIA(2), respectively, transfer blanks to the other 78 array elements of `IA`, and consider the read complete.

7.3 SEQUENTIAL INPUT/OUTPUT STATEMENTS

There are two sequential input/output statements: READ and WRITE. The READ and WRITE statements cause transfer of records.

7.3.1 READ Statements

These statements are used to obtain data values from an external source. The data values are input in either formatted or unformatted mode. The general form of the READ statement is:

READ (a,b,ERR = c,END = d) list

where

a — is a FORTRAN unit number

b — is optional; and, is either the statement label of the FORMAT statement describing the record(s) being read, or the name of an array containing a format specification.

ERR = c is optional; and, c is the label of a statement in the same program unit as the READ statement to which transfer is made if a transmission error occurs during data transfer.

END = d is optional; and, d is the label of a statement in the same program unit as the READ statement to which transfer is made upon encountering the end of the file.

list is optional; and, is an I/O list.

The value of a must always be specified, but under appropriate conditions b, c, d, and list can be omitted. The order of the parameters ERR = c and END = d can be reversed within the parentheses.

Transfer is made to the statement specified by the ERR parameter if an input error occurs. No indication is given of which record or records could not be read, only that an error occurred during transmission of data. If the ERR parameter is omitted and an error occurs, object program execution is terminated on the next I/O statement (unless the IOCHK subprogram described in section 7.8 is used).

Transfer is made to the statement specified by the END parameter when the end of the file is encountered; i.e., when a READ statement is executed after the last record on the file has already been read. No indication is given of the number of list items read into before the end of the file was encountered. If the END parameter is omitted and an end-of-file is encountered, object program execution is terminated on the next I/O statement.

The basic forms of the sequential READ statements are:

Form	Purpose
READ (a,b,ERR = c, END = d) list	Formatted READ
READ (a,ERR = c, END = d) list	Unformatted READ

Unformatted I/O files are not structured, except for that imposed by the physical record size associated with the FORTRAN unit number referencing them. Thus, they may be manipulated by other VORTEX processors.

Formatted READ

The form READ (a,b) list is used to read data from the file associated with FORTRAN unit number a into the variables whose names are given in the list. The data is transmitted from the file to storage according to the specifications in the FORMAT statement, which is statement label b.

Example

```
READ (5,98) A,B,(C(I,K),I=1,10)
```

Explanation

The above statement causes input data to be read from the file associated with FORTRAN unit number 5 into the variables A, B, C(1,K), C(2,K),..., C(10,K) in the format specified by the FORMAT statement whose statement number is 98.

Unformatted READ

The form READ(a) list is used to read a single record from the file associated with FORTRAN unit number a into the variables whose names are given in the list. Since the data is unformatted, no FORMAT statement label is given. This statement may be used to read unformatted data written by a WRITE(a) list statement. If the list is omitted, a record is passed over without being read.

Example

```
READ (J) A,B,C
```

Explanation

The above statement causes data to be read from the file associated with file reference number J into the variables A, B, and C.

7.3.2 WRITE Statements

WRITE statements are used to transfer program data to external devices. These data may be formatted or unformatted. The general form of the *WRITE* statement is:

WRITE (a,b,ERR = c,END = d) list

where

a	is an unsigned integer constant or an integer variable and represents a FORTRAN unit number.
b	is optional; and, is either the statement label of the FORMAT statement describing the record(s) being written, or the name of an array containing a format specification.
ERR = c	is optional; and, c is the label of a statement in the same program unit as the WRITE statement to which transfer is made if a transmission error occurs during a data transfer.

END = d is optional; and, d is the label of a statement in the same program unit as the WRITE statement to which transfer is made upon encountering the end of the file or device.

list is optional; and, is an I/O list.

The value of a must always be specified, but under appropriate conditions b, c, d, and list can be omitted. The order of the parameters ERR = c and END = d can be reversed within the parentheses.

Transfer is made to the statement specified by the ERR parameter if an output error occurs. No indication is given of which record or records could not be written, only that an error occurred during transmission of data. If the ERR parameter is omitted and an error occurs, object program execution is terminated on the next I/O statement.

Transfer is made to the statement specified by the END parameter when an end of the file or device is encountered.

The basic forms of the WRITE statement are:

Form	Purpose
WRITE (a,b,ERR = c, END = d) list	Formatted WRITE
WRITE (a,ERR = c, END = d) list	Unformatted WRITE

Formatted WRITE

The form WRITE(a,b) list is used to write data into the file whose reference number is from the variables whose names are given in the list. The data is transmitted from storage to the file according to the specifications in the FORMAT statement whose statement label is b.

Example

```
WRITE(7,75) A,(B(I,3),I=1,10,2),C
```

Explanation

The above statement causes data to be written from the variables A, B(1,3), B(3,3), B(5,3), B(7,3), B(9,3), C into the file associated with FORTRAN unit number 7 in the format specified by the FORMAT statement whose statement label is 75.

Unformatted WRITE

The form WRITE(a) list is used to write a single record from the variables whose names are given in the list into the file whose FORTRAN unit number is a. This data can be read back into storage with the unformatted form of the READ statement, READ(a) list.

Example

```
WRITE (L)((A(I,J),I=1,10,2),B(J,3),J=1,K)
```

Explanation

The above statement causes data to be written from the variables A(1,1), A(3,1),..., A(9,1), B(1,3), A(1,2), A(3,2),..., A(9,2), B(2,3),..., B(K,3) into the file associated with the file reference number L. Since the record is unformatted, no FORMAT statement number is given. Therefore, no FORMAT statement number should be given in the READ statement used to read the data back into storage.

7.4 DIRECT-ACCESS INPUT/OUTPUT STATEMENTS

The direct-access statements permit a programmer to read and write records randomly from any location within a file. They contrast with the sequential input/output statements, described previously, that process records, one after the other, from the beginning of a file to its end. With the direct-access statements, a programmer can go directly to any point in the file, process a record, and go directly to any other point without having to process all the records in between.

There are four direct-access input/output statements: READ, WRITE, DEFINE FILE, and FIND. The READ and WRITE statements cause transfer of data into or out of internal storage. These statements allow the user to specify the location within a file from which data is to be read or into which data is to be written.

The DEFINE FILE statement describes the characteristics of the file(s) to be used during a direct-access operation. The FIND statement updates the associated variable. In addition to these four statements, the FORMAT statement (described previously) specifies the form in which data is to be transmitted. The direct-access READ and WRITE statements and the FIND statement are the only input/output statements that may refer to a FORTRAN unit number defined by a DEFINE FILE statement.

Each record in a direct-access file has a unique record number associated with it. The programmer must specify in the READ, WRITE, and FIND statements not only the FORTRAN unit number, as for sequential input/output statements, but also the number of the record to be read, written, or found. Specifying the record number permits operations to be performed on selected records of the file, instead of on records in their sequential order.

The number of the record physically following the one just processed is made available to the program in an integer variable known as the associated variable. Thus, if the associated variable is used in a READ or WRITE statement to specify the record number, sequential processing is automatically secured. The associated variable is specified in the DEFINE FILE statement, which also gives the number, size, and type of the records in the direct-access file.

7.4.1 Define File Statement

Each direct-access file must be described once, in either the main program or a subprogram. The DEFINE FILE statement must logically precede any input/output statement referring to the file being described. The first DEFINE FILE statement encountered for a file is the one used during program execution. Subsequent descriptions are ignored. The statement has the general form

DEFINE FILE a1(m1,r1,f1,v1),a2(m2,r2,f2,v2),...

where

ai — represents an unsigned integer constant that is the FORTRAN unit number. a is also used to represent the file name within the partition indicated by the associated variable. The file name is of the form:

FILEnn

where nn = ai $0 \leq nn \leq 99$. A maximum of 10 such files is allowed.

mi — represents an integer constant that specifies the number of records in the file associated with ai.

ri — represents an integer constant that specifies the maximum size of each record associated with ai. The record size is measured in characters (bytes), or storage units (words). The method used to measure the record size depends upon the specification for fi

fi — specifies that the file is to be read or written either with or without format control; fi may be one of the following letters:

L indicates that the file is to be read or written either with or without format control, and that the maximum record size is measured in number of bytes.

E indicates that the file is to be read or written with format control (as specified by a FORMAT statement), and that the maximum record size is measured in number of characters (bytes).

U indicates that the file is to be read or written without format control, and that the maximum record size is measured in number of words.

v represents a single word integer variable (not an array element) called an associated variable. Prior to the DEFINE FILE statement, v must be set to contain the LUN of the partition and the protection key for the indicated partition. The LUN is a binary value from 0 to 255 and occurs in the right byte. The protection key is an ASCII graphic and occurs in the left byte. At the conclusion of each read or write operation, v is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a FIND operation, v is set to a value that points to the record found.

The associated variable cannot appear in the I/O list of a READ or WRITE statement for a file associated with the DEFINE FILE statement.

Example

```
DATA I2,J3/ZD014,ZA1
DEFINE FILE 8(50,100,L,I2),9(100,50,L,J3)
```

This DEFINE FILE statement describes two files (FILE08 and FILE09), referred to by unit reference numbers 8 and 9. The data in the first file consists of 50 records, each with a maximum length of 100 storage locations. The L specifies that the data is to be transmitted either with or without format control. I2 is the associated variable that serves as a pointer to the next record. The partition is logical unit number 20 with protection key "P".

The data in the second file consists of 100 records, each with a maximum length of 50 storage locations. The L specifies that the data is to be transmitted either with or without format control. J3 is the associated variable that serves as a pointer to the next record. The partition is logical unit number 161 and since the protection key is not an ASCII graphic, VORTEX will expect the partition to be unprotected.

7.4.2 Direct-Access READ Statement

The direct-access READ statement causes data to be transferred from a direct-access device into internal storage. The file being read must be defined with a DEFINE FILE statement. It has the general form

READ (a'r,b,ERR = c,END = d)list

where

a — is a FORTRAN unit number.

r — is an integer expression that represents the relative position of a record within the file associated with a.

b — is optional; and, if given, is either the statement label of the FORMAT statement that describes the data being read or the name of an array that contains an object-time format specification.

ERR = c — is optional; and, c is the label of a statement in the same program unit as the READ statement to which control is given when a device error condition is encountered during data transfer from device to storage.

END = d — is optional, and is the label of a statement in the same program unit as the READ statement to which transfer is to be made upon encountering an end-of-file.

list — is optional; and, is an I/O list.

The I/O list must not contain the associated variable defined in the DEFINE FILE statement for unit a.

The relative record number of the first record of a direct-access file is 1.

7.4.3 Direct-Access WRITE Statement

The direct-access WRITE statement causes data to be transferred from internal storage to a direct-access device. The file being written must be defined with a DEFINE FILE statement. The statement has the general form:

WRITE (a'r,b,ERR = c,END = d) list

where

a — is a FORTRAN unit number.

r — is an integer expression that represents the relative position of a record within the file associated with a.

b	is optional; and, if given, is either the statement label of the FORMAT statement that describes the data being written or the name of an array that contains an object-time format.
ERR = c	is optional; and, c is the label of a statement in the same program unit as the READ statement to which control is given when a device error condition is encountered during data transfer from device to storage.
END = d	is optional, and is the label of a statement in the same program unit as the READ statement to which transfer is to be made upon encountering an end-of-file.
list	is optional; and, is an I/O list.

The I/O list must not contain the associated variable defined in the DEFINE FILE statement for unit a.

7.4.4 FIND Statement

The FIND statement is included for compatibility with other processors. Its syntax is checked, but it performs no hardware function at execution time. The associated variable is, however, updated. The statement has the general form:

```
FIND (a'r)
```

where

a	is a FORTRAN unit number.
r	is an integer expression that represents the relative position of a record within the file associated with a.

The file on which the record is being found must be defined with a DEFINE FILE statement.

Example

```
      DEFINE FILE 8(1000,80,L,IVAR)
10    FIND (8'50)
        .
        .
15    READ (8'50) A,B
```

While the statements between statements 10 and 15 are executed, record 50, in the file associated with unit number 8, is found. After the FIND statement is executed, the value of IVAR is 50. After the READ statement is executed, the value is 51.

7.5 FORMAT STATEMENTS

FORMAT statements, with input/output operations, specify conversion and editing of information between program storage and external representation. *FORMAT* statements are nonexecutable and must have a statement label to be referenced by input/output statements. Conversion performed according to a *FORMAT* statement during output is in general the reverse of conversion performed during an input operation.

A *FORMAT* statement is expressed as:

n FORMAT (21t1z1t2z2...tnznq2)

where

n	is the statement label.
qi	is a series of slashes or is empty.
zi	is a field separator.
ti	is a field descriptor or group of field descriptors.

The noun *FORMAT* and the parentheses must appear in this form.

The following list gives general rules for using the FORMAT statement:

a. FORMAT statements are not executed; their function is to supply information to the object program. However, they must be placed among the executable statements, and cannot occur in a BLOCKDATA subprogram.

b. Complex data items in records are processed exactly like two consecutive real items.

c. Any number of commas or slashes can be used as separators between format codes.

d. When defining a FORTRAN record by a FORMAT statement, it is important to consider the maximum size record allowed on the input/output medium. FORTRAN will read or write multiple records if move data then will fit in a record is requested.

e. When formatted records are prepared for printing at a printer or terminal, the first character of the record is not printed. It is treated as a carriage control character:

Character	Meaning
blank	Advance one line before printing
0	Advance two lines before printing
1	Advance to first line of next page
+	No advance (overprint)

For media other than a printer or terminal, the first character of the record is treated as data. Refer to the VORTEX reference manuals for devices that process the + character.

f. If the I/O list is omitted from the READ or WRITE statement, a record is skipped on input, or a blank record is inserted on output, unless the record was transmitted between the file and the FORMAT statement (see Hollerith format descriptors).

Various Forms of a FORMAT Statement

All of the field descriptors in a FORMAT statement are enclosed in a pair of parentheses. Within these parentheses, the format codes are delimited by the separators, slash and comma. The slash indicates the end of the physical record; the comma indicates the end of a data item within the record.

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information provided jointly by the I/O list, if one exists, and the format specification. There is no I/O list item corresponding to the format descriptors T, X, or Hollerith. These communicate information directly with the record.

Whenever an I, D, E, F, G, A, L, or Z code is encountered, format control determines whether there is a corresponding element in the I/O list. If there is such an element, appropriately converted information is transmitted. If there is no corresponding element, the format control terminates, even if there is an unsatisfied repeat count.

If, however, format control reaches the last (outer) right parenthesis of the format specification, a test is made to determine if another element is specified in the I/O list. If not, control terminates. However, if another list element is specified, the format control demands that a new record start. Control therefore reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification.

Given the following FORMAT statements:

```
70 FORMAT (2(I3,F5.2),I4,F3.1)
80 FORMAT (I3,F5.2,2(I3,2F3.1))
90 FORMAT (I3,F5.2,2I4,5F3.1)
```

With additional elements in the I/O list after control has reached the last right parenthesis of each, control would revert to the 2 (I3,F5.2) specification in the case of statement 70; to 2(I3,2F3.1) in the case of statement 80; and to the beginning of the format specification, I3,F5.2,... in the case of statement 90.

The question of whether there are further elements in the I/O list is asked only when an I, D, E, F, G, A, L, or Z code or the final right parenthesis of the format specification is encountered. Before this is done, T, X, and Hollerith codes, and slashes are processed. If there are fewer elements in the I/O list than there are format codes, the remaining format codes are ignored.

Comma: The simplest form of a FORMAT statement is the one shown in the general form at the beginning of this section. The format codes, separated by commas, are enclosed in a pair of parentheses. One FORTRAN record is defined within a single pair of left and right parentheses. The following examples illustrate the use of the format codes I, F, D, E, Z, and G.

Example

```
75 FORMAT (I3,F5.2,E10.3,G10.3)

   READ (5,75) N,A,B,C
```

Explanation

a. For input fields are described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the READ statement is executed, one input card is read from the file associated with FORTRAN unit number 5.

b. When an input card is read, the number in the first field of the card (three columns) is stored in integer format in location N. The number in the second field of the input card (five columns) is stored in real format in location A, etc.

c. If there were one more variable in the I/O list, say M, another card would be read and the information in the first three columns of that card would be stored in integer format in location M. The rest of the card would be ignored.

d. If there were one fewer variable in the list (say C is omitted), format specification G10.3 would be ignored.

Slash: A slash is used to indicate the end of a FORTRAN record format. For example, the statement:

```
25    FORMAT     (I3,F6.2/D10.3,F6.2)
```

describes two FORTRAN record formats. The first, third, etc., records are transmitted according to the format I3, F6.2 and the second, fourth, etc., records are transmitted according to the format D10.3,F6.2.

Consective slashes can be used to introduce blank output records or to skip input records. If there are n consecutive slashes at the beginning or end of a FORMAT statement, n input records are skipped or n blank records are inserted between output records. If n consecutive slashes appear anywhere else in a FORMAT statement, the number of records skipped or blank records inserted is n-1. For example, the statement:

```
25    FORMAT    (1X,10I5//1X,8E14.5)
```

describes three FORTRAN record formats. On output, it causes double spacing between the line written with format 1X,10I5 and the line written with the format 1X,8E14.5.

7.5.1 FIELD Descriptors

FIELD descriptors describe the type of conversion and editing to be performed on each variable appearing in the input/output list. *FIELD* descriptors can be in any of the following forms:

rAw rFw.d rEw.d rDw.d rIw nHs 's' nX rLw rGw.d Ty Zh

a. The characters A, D, E, F, G, L, Z, and I indicate the manner of conversion for variables in the list.

b. The character H, and ' ' indicate that characters are to be input/output directly from the format.

c. The character / represents the end of a record.

d. w and n are non-zero integer constants defining the width of the field (including digits, decimal point, and algebraic signs) in the external character string.

e. d is an integer specifying the number of fractional digits appearing in the external string.

f. r is an optional, non-zero integer indicating that the specification is to be repeated r times.

g. s is a string of acceptable FORTRAN characters.

h. The T descriptor relocates the current absolute position in the external record.

i. The X descriptor relocates the current relative position in the external record.

j. y is a non-zero integer constant specifying the character position in the external record.

k. Z format code is used in the transmitting hexadecimal data.

l. h denotes a string of hexadecimal digits.

7.5.2 A Format Code

An A format code is used in conjunction with a *READ* or *WRITE* statement for the input/output of alphanumeric information to ro from a list element. The general form is

rAw

where r and w are unsigned integer constants. If r is one, it can be omitted.

Input: rAw will be interpreted to mean that the next r successive fields of w characters are each to be stored in the associated *REAL* list elements. If w is greater than c, where c is the number of characters a single list element can contain, only the c right-most characters will be significant. If w is c or less, the characters will be left-justified, and the word(s) filled with blanks, if necessary.

Output: rAw will be interpreted to mean that the next r successive fields of w characters are each to be the result of alphanumeric transmission from the specified list elements. If w exceeds g, only g characters of output will be transmitted, preceded by w - g blanks. If w is g or less, the w left-most characters of the specified storage element will be transmitted.

where g is the number of characters a single list element can contain.

7.5.3 D Format Code

The *D* is used for the input/output of double-precision numbers. It is used exactly as the *E* except the letter E is replaced by D.

7.5.4 E Format Code

The E format is used in transmitting real data. The data must not exceed the maximum magnitude for a real constant. The general form is

rEw.d

Input: Each external value is of field width w with d characters in the fractional part of the value. The value is right-justified with all blanks counting as zeros. A minus sign may precede the value of the exponent. A decimal point placed in the fractional part takes precedence over the d specification. The character E may be present to separate the value and the exponent.

For a field specification of E10.3:

123E3	is converted to	123.0
12874E2	is converted to	1287.4
-563E-02	is converted to	-0.00563
398E00	is converted to	0.398
5387601	is converted to	538.7601
5455-01	is converted to	0.5455
-6.7563E05	is converted to	-675630.0

Output: Internal values are converted to decimal values of the forms:

.ddd...dE+ee and .ddd...E-ee

where

ddd...d represents d digits, and ee is a decimal exponent.

The leading decimal point and E characters are present exactly as shown. Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right-justified and preceded by blanks to fill the width, w. This field width includes the exponent digits, the sign of the expondnet (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). This means that the field width should correspond to the relation: $w \geq d + 6$.

If w is too small, the output will be truncated to w-1 characters, and an asterisk (*) placed in the last character position to flag the error.

For a field specification of E12.5:

76.573	is converted to	.76573E 02
58796.341	is converted to	.58796E 05
-369.7583	is converted to	-.36976E 03
0.006873	is converted to	.68730E -02
0.2	is converted to	.20000E 00
-0.0000054	is converted to	-.54000E -05

7.5.5 F Format Code

The F format code is used in transmitting real data. The data must not exceed the maximum magnitude for a real constant. The general form is

rFw.d

Input: Input strings are decimal numbers of length w with d characters in the fractional portion. Blanks are treated as zeros. If a decimal point is present in a value, the fractional portion of the value is explicitly defined by that decimal point character.

For a field specification F8.3:

35	is converted to	0.035
964372	is converted to	964.372
0.53821	is converted to	0.53821
-16.402	is converted to	-16.402
-12	is converted to	-0.012
47.-4	is converted to	0.0047

Output: The field is right-justified with as many leading blanks as necessary to fill w. Negative values are preceded by a minus sign. Internal values are converted to fixed-point decimal numbers and rounded to d decimal places.

For a field specification of F10.4:

368.4	is converted to	368.4000
12.0	is converted to	12.0000
-17.90767	is converted to	-17.9077
-37.5E-2	is converted to	.3750

If a value requires more positions than allowed by w, the most significant digits, including sign if negative, are output. The error indication is designated by an asterisk in the least significant character position.

For a field specification of F6.4:

4739.76	is converted to	4740*
-12.463	is converted to	-12.5*

7.5.6 G Format Code

The G format code is a generalized code in that it automatically selects an output format appropriate to the magnitude of the real data. The general form is

rGw.d

where

r is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

w is an unsigned integer constant specifying the total field length.

d is an unsigned integer constant specifying the number of significant digits.

Input: Input processing is the same as for the *F* conversion.

Output: The output format of an item whose magnitude is N, is as follows:

Magnitude of N	**Format**
$0.1 \leq N < 1$	F(w-4).d,4x
$1 \leq N < 10$	F(w-4).(d-1),4x
.	.
.	.
.	.
$10^{d-1} < N < 10^{d}$	F(w-d).0,4x
otherwise	sEw.d (s is scale factor)

For the purpose of simplification, the following examples deal with the printed line. However, the concepts apply to all input/output media.

Example 1

Assume that the variables A, B, C, and D are of type real whose values are 292.7041, 82.43441, 136.7632, 0.8081945, respectively.

```
1     FORMAT    (G12.4,G12.5,G12.4,G12.7)
2     FORMAT    (G13.4,G13.5,G13.4)
3     FORMAT    (G13.4)
        .
        .
        .
      WRITE     (5,n)A,B,C,D
        .
        .
        .
```

Explanation

a. If n has been specified as 1, the printed output would be as follows (b represents a blank):

```
Print Position 1                                    Print Position 48
'                                                   '
bbb292.7bbbbbb82.434bbbbbbb136.8bbbb.8081945bbbb
```

b. If n has been specified as 2, the printed output would be:

```
Print Position 1                                    Print Postion 39
'                                                   '
bbbb292.7bbbbbbb82.434bbbbbbbb136.8bbbb                          Line 1
bbbb.8082bbbb                                                    Line 2
```

From the above example, it can be seen that by increasing the field width reserved (w), blanks are inserted.

c. If n has been specified as 3, the printed output would be:

```
Print Position 1
'
bbbb292.7bbbb                                                    Line 1
bbbb82.43bbbb                                                    Line 2
bbbb136.8bbbb                                                    Line 3
bbbb.8082bbbb                                                    Line 4
```

From the above example, it can be seen that the same format code is used for each variable in the list. Each repetition of the same format code causes a new line to be printed.

7.5.7 Hollerith Field Descriptor

In FORTRAN, Hollerith information consists of the legal FORTRAN character set plus the additional characters

" # : ; ¢ % & ' ! [ ] < > ?

Information input from the typewriter or paper tape is converted to the internal ASCII code used by FORTRAN. When this information is output, the internal codes are converted to the appropriate typewriter or paper tape codes. The general form is

nHs

or:

's'

Input: The w characters in the string, s, are replaced by the next w characters from the input record. The result is a new string in the field specification. Each apostrophe in a pair is overlayed by an input character in the 's' format.

Example

Specification	Input String	Resultant Specification
5H12345	ABCDE	5HABCDE
7HbTRUEbb	FALSEbb	7HFALSEbb
8Hbbbbbbbb	MATRIXbb	8HMATRIXbb
'AB'	12	'12'
'X'	ABC	'ABC'

b indicates a blank space

This feature can be used to change titles, dates, headings, etc., that are output with the program data.

Output: The number of characters, n, in the string, s, should contain exactly the number of characters specified so that characters from other fields are not taken as part of the string.

Blanks are countered as characters in the string. The apostrophe character (') can be output using a pair of them in the 's' format description.

Example

Specification	External Output
1HR	R
8HbSTRINGb	bSTRINGb
11HX(1,3)=12.0	X(1,3)=12.0
'bA='	bA=
'bs=''A'''	bs='A'

b indicates a blank space

7.5.8 I Format Code

Only integer data may be processed by the I format code. The general form is

rIw

Input: External input values are right-justified with the width, w. Blanks are counted as zeros. Input values must be integer values. A preceding minus sign may be placed on a value.

For a field specification of I4:

120	is converted to	120
-144	is converted to	-144
102	is converted to	102
-3	is converted to	-3

Output: Internal values are converted to integer constants. Negative values are preceded by a minus sign. Each field is right-justified and filled with leading blanks.

For a field specification of I6:

281	is converted to	281
-3567	is converted to	-3567

If the data require more character positions than allowed by the width, w, only the most significant w positions are output.

For a field specification of I3:

2810	is converted to	3*
-6374	is converted to	-6*

7.5.9 L Format Code

The L format code is used in transmitting logical variables. The general form is

rLw

where

r	is optional; and, is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.
w	is an unsigned integer constant that specifies the number of characters of data.

Logical variables may be read or written by means of the format code Lw.

Input: The first T or F encountered in the next w characters of the input record causes a value of .TRUE. or .FALSE., respectively, to be assigned to the corresponding logical variable. If field w consists entirely of blanks, a value of .FALSE. is assumed.

Output: A T or F is inserted in the output record as the value of the logical variable in the I/O list. T is a non-zero value and F is zero. The single character is preceded by w - 1 blanks.

7.5.10 T Format Code

The T format code specifies the absolute position in the buffer where the buffer pointer is to be positioned. The general form is

Ty

where

T	is a descriptor that relocates the current position in the buffer.
y	is a non-zero positive integer constant that specifies the character position in the buffer.

Input: A T specification can be used to skip or re-read fields.

Output: A T specification can be used to position column headers as follows (b indicates a blank space):

```
1     FORMAT(T10,5HCOLb1,T22,5HCOLb2)
```

This example causes

```
COLb1
```

to be printed starting in column 10, and causes

```
COLb2
```

to be printed starting in column 22.

7.5.11 X Format Code

The X format code specifies the relative position in the buffer where the buffer pointer is to be positioned. Positioning is always forward, with blank fill on WRITE or ENCODE. The general form is

nX

where

n is the number of characters skipped or filled ($n > 0$).

Input: n spaces are skipped from the input record.

Example

Specification	Input String	Resultant Input
F4.1, 3X, F3.0	12.5RRR120	12.5,120.

The RRR characters are ignored by the 3X specification.

Output: n blanks are inserted in the external record.

Example

Specification	Output
1HA, 4X, 2HBC	AbbbbBC
4X, 3HABC	bbbbABC
1X, ABC, 3X	bABCbbb

7.5.12 Z Format Code

The hexadecimal Z format code causes a string of hexadecimal digits to be interpreted as a hexadecimal value and to be associated with the corresponding I/O list element for purposes of data transmitting. It has the general form:

Zw

where

w denotes a string of hexadecimal digits. The maximum value that can be read depends on the number of words in the corresponding item in the I/O list.

Input: Scanning of the input field proceeds from right to left. Leading, embedded, and trailing blanks in the field are treated as zeros. One word in internal storage contains four hexadecimal digits; thus, if an input field contains a digit count that is not a multiple of four, the number will be padded on the left with hexadecimal zeros when it is stored. If the storage area is too small for the input data, the data is truncated and high-order digits are lost.

Output: If the number of characters in the storage location is less than w, the left-most print positions are filled with blanks. If the number of characters in the storage location is greater than w, the left-most digits are truncated and the rest of the number is printed.

7.5.13 Scale Factor P

The representation of the data, internally or externally, can be modified by the use of a scale factor followed by the letter P preceding the F, E, G, and D format codes.

The scale factor affects the appropriate conversions in the following manner:

a. For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:
externally represented number equals internally represented number times the quantity ten raised to the nth power.

b. For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

c. For E and D output, the basic real constant part of the quantity is multiplied by ten to the nth power and the exponent is reduced by the scale factor.

d. For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

For example, if input data are in the form xx.xxxx and it is desired to use this internally in the form .xxxxxx, the format code used to effect this change is 2PF7.4.

Input: As another example, consider the following input data:

27bb-93.2094bb-175.8041bbbb55.3647

where b represents a blank.

The following statements:

```
5     FORMAT          (I2,3F11.4)
        .
        .
        .
      READ            (4,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K :  27               B :  -175.8041
A :  -93.2094         C :  55.3647
```

The following statements:

```
5     FORMAT          (I2,1P3F11.4)
        .
        .
        .
      READ            (4,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K :  27               B :  -17.58041
A :  -9.32094         C :  5.53647
```

The following statements;:

```
5     FORMAT          (I2,1P3F11.4)
        .
        .
        .
      READ            (4,5) K,A,B,C
```

causes the variables in the list to assume the following values:

```
K :  27               B :  -1758.041
A :  -932.094         C :  553.647
```

Output: Assume the variables K,A,B, and C have the following values:

```
K :  27               B :  -175.8041
A :  -93.2094         C :  55.3647
```

then the following statements:

```
5     FORMAT          (I2,1P3F11.4)
        .
        .
        .
      WRITE           (5,5) K,A,B,C
```

cause the variables in the list to output the following values:

```
K :  27               B :  -1758.041
A :  -932.094         C :  553.647
```

The following statements:

```
5     FORMAT          (I2,-1P3F11.4)
        .
        .
        .
      WRITE           (5,5) K,A,B,C
```

cause the variables in the list to output the following values:

```
K :  27              B :  -17.5804
A :  -9.3209         C :  5.5365
```

For output, when scale factors are used, they have effect only one real data. However, this real data may contain an E or D decimal exponent. A positive scale factor used with real data that contains an E or D decimal exponent increases the number and decreases the exponent. Thus, if four data items in a I/O list were 27, .9321E02, .1758E03, and .3536E02, and the statement FORMAT (1X,I2,3E13.3) is used with an appropriate *WRITE* statement, the following printed line is output:

```
b27bbbb-.932Eb02bbbb-.175Eb03bbbbb.553Eb02
```

the statement FORMAT (1X,12,1P3E13.3) used with the same *WRITE* statement results in the following printed output:

```
b27bbb-9.321Eb01bbb-1.758Eb02bbbb5.536Eb01
```

The statement FORMAT (1X,12,-1P3E13.3) used with the same WRITE statement results in the following printed output:

```
27bbbb-.093Eb03bbbb-.018Eb04bbbbb.055EB03
```

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all format codes following the scale factor within the same *FORMAT* statement. This also applies to format codes enclosed within an additional pair of parentheses.

7.6 AUXILIARY I/O STATEMENTS

Auxiliary I/O statements (ENDFILE, REWIND, and BACKSPACE) do not apply to direct access files (i.e., those files opened by a DEFINE FILE statement.

7.6.1 ENDFILE Statement

The ENDFILE statement defines the end of the file associated with a FORTRAN unit number. On non-mass-storage files it cause an end-fo-file record to be written. On mass storage files, it will cause a CLOSE/update followed by an OPEN/Leave. The general form is

```
ENDFILE a
```

where

a is a FORTRAN unit number.

7.6.2 REWIND Statement

The REWIND statement repositions a file associated with FORTRAN unit number (a), causing a subsequent READ or WRITE statement referring to a to read data from or write data into the first record of the file associated with a.

```
REWIND a
```

where

a is a FORTRAN unit number

7.6.3 BACKSPACE Statement

The BACKSPACE statement causes the file associated with a to backspace one record. If the file associated with a is already at its beginning, or if the device (e.g., card punch) does not permit it, execution of this statement has no effect. This statement may not be executed for direct-access files. The general form is

```
BACKSPACE a
```

where

a is a FORTRAN unit number.

7.7 ENCODE/DECODE STATEMENTS

This section explains the use of the ENCODE and DECODE statements.

7.7.1 ENCODE Statement

The ENCODE statement takes an I/O list, converts each element and places it in a specified buffer. This statement performs data conversion according to a FORMAT statement without performing external I/O operations. The general form is

```
ENCODE (c, f, a, i) list
```

where

c is the number of characters in the record.

f is the format statement label or array name.

a is the name of an array to be buffered.

i is optional, and, is the integer variable into which the number of characters processed will be stored.

list is the input/output list.

7.7.2 DECODE Statement

The DECODE statement works from the buffer into the I/O list. This statement performs data conversion according to a FORMAT statement without performing external I/O operations. The general form is

DECODE (a, f, a, i) list

where

c is the number of characters in the record.

f is the format statement label or array name.

a is the name of an array to be buffered.

i is optional, and, is the integer variable into which the number of characters processed will be stored.

list is the input/output list.

Example

```
      DIMENSION I(40)
      READ(CDR,10)I
10    FORMAT(40A2)
      DECODE(10,20,I)K,L
20    FORMAT(2I5)
```

These statements read an ASCII card image into array I. The first two fields of five ASCII characters are then decoded into their integer equivalent and placed into the variables K and L.

Note: If too many characters are generated--the extras will be lost. If not enough characters are generated, the remainder of the buffer is filled with blanks.

7.8 IOCHK

The IOCHK subprogram provides an additional method for detecting end and error conditions. It can be referenced as a subroutine or a function. IOCHK has one parameter (I), whose value is set after a READ or WRITE, according to the following table:

I	Meaning
0	Normal I/O completion
-1	End-of-file or end-of-device
1	I/O error

Example (subroutine)

```
READ(u)...
CALL IOCHK(I)
IF (I) 1,2,3
```

where 1 = EOF exit
2 = Normal exit
3 = Error ext

Example (Function)

```
    READ(u)
    IF(IOCHK(I).EQ.0) GO TO 2
    IF (I.LT.0) GO TO 3
  1 ...
```

where exits have same meaning as above.

SECTION 8
PROGRAMS AND SUBPROGRAMS

This section explains the use and structure of FORTRAN programs and subprograms.

8.1 PROGRAM COMPONENTS

FORTRAN programs consist of program parts, program bodies, TITLE statements, Subprogram statements, and NAME statements. Comment lines may be interspersed arbitrarily among them.

8.1.1 Program Part

A program part must include at least one executable statement, and may include any number of FORMAT statements. Optionally, this collection may be preceded by statement function definitions and DATA statements.

8.1.2 Program Body

A program body is a (possibly empty) collection of specification statements, followed by a program part, followed by an END line.

8.1.3 TITLE Statement

Each VORTEX program or subprogram can contain as its first statement (except for comment lines) a TITLE statement with the following format:

```
TITLE n
```

where

n	is a character string representing the program module name that is included in the heading of the source listing, as well as in the object program. This name is used by system maintenance and generation programs in VORTEX.

8.1.4 Subprogram Statements

Following are the three types of FORTRAN Subprogram statements:

- FUNCTION
- SUBROUTINE
- BLOCK DATA

8.1.5 NAME Statement

A main program can accept a main program entry name definition of the following format:

NAME N1, N2, ..., Nn

where

N1, N2, ..., Nn are entry names by which the main program can be referenced

8.2 MAIN PROGRAMS

A main program consists of a program body that is optionally preceded by NAME statements.

A main program cannot contain a subprogram definition statement, namely:

a FUNCTION statement
a SUBROUTINE statement
a BLOCK DATA statement

A main program may contain calls to other subprograms or may contain statement function subprograms.

A VORTEX main program must include specifications for all common blocks that are referenced by the subprograms.

8.3 SUBPROGRAMS

A subprogram consists of a FUNCTION or SUBROUTINE statements, optionally preceded by a TITLE statement and/or comment lines, followed by a program body, or is a BLOCK DATA subprogram.

Subprograms are program units which may be called by other programs or subprograms. Subprograms are categorized as one of the following:

PROCEDURE SUBPROGRAMS
    FUNCTION subprogram
    SUBROUTINE subprogram

SPECIFICATION subprogram
    BLOCK DATA subprogram

Functions are programmed procedures that are often used to provide solutions t. mathematical functions. Function references may be used in the same manner as references to variables in an expression. For example: X = AB*SIN (Y) - C*COS (Y*Z), where SIN is the name of the sine function, COS is the name of the cosine function, and (Y) and (Y*Z) are their respective argument lists. The value returned for a function reference is of the same mode as the function name, corresponding to the rules for real and integer symbolic names.

A subprogram name consists of from one to six alphameric characters, the first of which must be alphabetic. A subprogram name may not contain special characters.

Type declaration of FUNCTION Subprograms may be made by the predefined convention, by the IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the FUNCTION subprogram. The type of a function determines the type of the result that can be returned from it.

No type is associated with a SUBROUTINE name because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the implicit arguments in COMMON.

8.3.1 Function Subprograms

The FUNCTION subprogram is a subprogram consisting of a FUNCTION statement followed by other statements including at least one RETURN statement. It is an independently written program that is executed wherever its name is referred to in another program. It has the general form

Type FUNCTION name*s(a1,a2,a3,...)

where

type — is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL. Its inclusion is optional.

name — is the name of the FUNCTION.

s — represents one of the permissible length specifications for its associated type. It may be included optionally only when Type is specified. It must not be used when DOUBLE PRECISION is specified.

a — is a dummy argument. It must be a distinct variable or array name (i.e., it may appear only once within the statement) or dummy name of a SUBROUTINE or other FUNCTION subprogram. There must be at least one argument in the argument list.

A type declaration for a function name may be made by the predefined convention, by an IMPLICIT statement, by an explicit specification in the FUNCTION statement, or by an explicit specification statement within the FUNCTION subprogram. The function name must also be typed in the program units which refer to it if the predefined convention is not used.

Since the FUNCTION is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

Excepting TITLE statements and comment lines, the FUNCTION statement must be the first statement in the subprogram. The FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, or a BLOCK DATA statement. If an IMPLICIT statement is used in a FUNCTION subprogram, it must immediately follow the FUNCTION statement.

Example 1

```
REAL FUNCTION SOMEF (A,B)
  .
  .
  .
SOMEF=A**2+B**2
  .
  .
  .
RETURN
END
```

Example 2

```
INTEGER FUNCTION CALC (X,Y,Z)
  .
  .
  .
CALC=X+Y+Z**2
  .
  .
  .
RETURN
END
```

Explanation

The *FUNCTION* subprograms SOMEF and CALC in Examples 1 and 2 are declared as type *REAL* and *INTEGER*, respectively.

8.3.2 Subroutine Subprograms

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects. The rules for naming FUNCTION and SUBROUTINE subprograms are similar. They both require an END statement, and they both contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE subprogram is a set of commonly used computations, but it need not return any results to the calling program, as does the FUNCTION subprogram. The SUBROUTINE subprogram is referenced by the CALL statement. It has the general from

```
SUBROUTINE name (a1,a2,a3,...)
```

where

name is the SUBROUTINE name

Each a is a distinct dummy argument (i.e., it may appear only once within the statement). There need not be any arguments, in which case the parentheses must be omitted. Each argument used must be a variable or array name, the dummy name of another SUBROUTINE or FUNCTION subprogram, or an asterisk, where the character "*" denotes a return point specified by a statement number in the calling program.

Since the SUBROUTINE is a separate program unit, there is no conflict if the variable names and statement numbers within it are the same as those in other program units.

The SUBROUTINE statement, except for TITLE statements and comment lines must be the first statement in the subprogram. The SUBROUTINE subprogram may contain any FORTRAN statement except a FUNCTION statement, another SUBROUTINE statement, or a BLOCK DATA statement. If an IMPLICIT statement is used in a SUBROUTINE subprogram, it must immediately follow the SUBROUTINE statement.

The actual arguments can be:

- A constant (including Hollerith constants)
- Any type of array name
- Any type of arithmetic or logical expression
- The name of a FUNCTION or SUBROUTINE subprogram
- A statement number preceded by "&"

Note: The last statement executed by a subroutine must be a *RETURN* statement.

Example

```
      SUBROUTINE R(A,I,Z)
      DIMENSION A(10)
      Z=0
      DO 1 J = 1,10
1     Z = Z+A(J)**I
      RETURN
      END
```

8.3.3 Multiple Entry into a Subprogram

The standard (normal) entry into a SUBROUTINE subprogram from the calling program is made by a CALL statement that refers to the subprogram name. The standard entry into a FUNCTION subprogram is made by a function reference in an arithmetic expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement.

It is also possible to enter a subprogram (either SUBROUTINE or FUNCTION) by a CALL statement or a function reference that references an ENTRY statement in the subprogram. Entry is made at the first executable statement following the ENTRY statement. It has the general form

ENTRY name (a1,a2,a3,...)

where

name	is the name of an entry point
a	is a dummy argument corresponding to an actual argument in a CALL statement or in a function reference.

An entry in a subroutine must be referred to by a CALL statement; an entry in a function must be referred to by a function reference.

ENTRY statements are non-executable and do not affect control sequencing during execution of a subprogram. A subprogram must not refer to itself directly or indirectly, or through any of its entry points. Entry cannot be made into the range of a DO. The appearance of an ENTRY statement does not alter the rule that statement functions in subprograms must precede the first executable statement of the subprogram.

Allowable dummy and actual arguments for an entry in a subroutine subprogram are the same as allowed for a subroutine subprogram. Allowable arguments for an entry in a function subprogram are the same as allowed for a function subprogram. A dummy argument in an ENTRY statement must also be a dummy argument in the subroutine or function statement; and, if it is an array, it must be dimensioned ogether with other specification statements at the beginning of the subprogram.

The dummy arguments in the ENTRY statement need not agree in order, type, or number within the dummy arguments in the SUBROUTINE or FUNCTION subprogram or any other ENTRY statement in the subprogram. However, the arguments for each CALL or function reference must agree in order, type, and number with the dummy agruments in the SUBROUTINE, FUNCTION, or ENTRY statement to which it refers.

Entry into a subprogram associates actual arguments with the dummy arguments of the referenced ENTRY statement. Thus, all appearances of these arguments in the whole subprogram become associated with actual arguments. A function reference, and hence any ENTRY statement in a FUNCTION subprogram, must have at least one argument.

A dummy argument must not be used in any executable statement in the subprogram unless it has been previously defined as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement.

If information for an object-time dimension array is passed in a reference to an ENTRY statement, the array name and all of its dimension parameters (except any that are in a common area) must appear in the argument list of the ENTRY statement.

In a FUNCTION subprogram, the types of the function name and entry name are determined by the predefined convention, by an IMPLICIT statement, by an explicit type-statement, or by a type in the FUNCTION statement. The types of these variables (i.e., the function name and entry names) can be different; the variables are treated as if they were equivalenced. After one of these variables is assigned a value in the subprogram, any others of different type become indeterminate in value.

When there is an ENTRY statement in a function subprogram, either the function name or one of the entry names must be assigned a value.

Upon exit from a FUNCTION subprogram, the value returned is the value last assigned to the function name or any entry name. It is returned as though it were assigned to the name in the current function reference. If the last value is assigned to a different entry name, and that entry name differs in type from the name in the current function reference, the value of the function is undefined.

8.3.4 Block Data Subprogram

To initialize variables in a *COMMON* block, a separate subprogram must be written. This separate subprogram contains only the *DATA, COMMON, DIMENSION, EQUIVALENCE,* and *TYPE* statements associated with the data being defined. This subprogram is not called; its presence suffices to provide initial data values for references in main and subprograms to labeled common blocks. Data may not be initialized in unlabeled common.

The general form is

```
BLOCK DATA
•
•
```

```
      .
      END
```

a. The *BLOCK DATA* subprogram may not contain any executable statements.

b. The *BLOCK DATA* statement, except for TITLE statements and comment lines must be the first statement in the subprogram.

c. All elements of a COMMON block must be listed in the *COMMON* statement, even though they are not all initialized; for example, the variable A in the *COMMON* statement in the following example does not appear in the data initialization statement.

```
BLOCK DATA
COMPLEX C
COMMON/ELN/C,A,B/RMG/Z,Y
DATA C/(2.4.3.769)/
```

d. Data may be entered into more than one COMMON block in a single *BLOCK DATA* subprogram.

e. An optional name n can follow the *BLOCK DATA* statement:

```
BLOCK DATA n
```

This causes output of n as an entry name so that the subprogram can be stored in a library enabling it to be loaded with any module containing an EXTERNAL n statement.

8.4 DATA STATEMENT

A DATA initialization statement is used to define initial values of variables, array elements, and arrays. There must be a one-to-one correspondence between the total number of elements specified or implied by the list k and the total number of constants specified by the corresponding list d after application of any replication factors, i.

For real, integer, complex, and logical types, each constant must agree in type with the variable or array element it is initializing. Any type of variable or array element may be initialized with a literal or hexadecimal constant.

This statement cannot precede any specification statement and it must precede all executable statements and statement function definitions. The DATA statement has the general form

```
DATA k1/d1/,k2/d2/,...
```

where

k — is a list containing variables, array elements (in which case the subscript quantities must be unsigned integer constants), array names, Dummy or implied DO-lists. Arguments may not appear in the list.

d — is a list of constants (integer, real, complex, hexadecimal, logical, or Hollerith), any of which may be preceded by i*. Each i is an unsigned integer constant. When the form i* appears before a constant, it indicates that the constant is to be specified i times.

Example 1

```
DIMENSION D(10)

DATA A,B,C/5.0,6.1,7.3/,D(1),D(2),D(3),D(4),D(5)/5*1.0/
```

Explanation

The *DATA* statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3, respectively. In addition, the statement specifies that the first five variables in array D are to be initialized to 1.0.

Example 2

```
DIMENSION A(5),B(3),L(2)

DATA A(1),A(2),A(3),A(4),A(5)/5*1.0/,B(1),B(2)/2*5.0/
 ,L(1),L(2)/.TRUE.,.FALSE
```

Explanation

The *DATA* statement specifies that all the variables in array A are to be initialized to 1.0 and the first two elements of array B are to be initialized to 5.0. The logical variables, (L(1) and L(2)), in array L are initialized to .TRUE. and .FALSE., respectively.

An initially defined variable, or any element, may not be in blank common. However, in a labeled *COMMON* block, they may be initially defined only in a block data subprogram. (See the Subprograms section.)

Example 3

```
DIMENSION A(3),B(3,2)
DATA A/1.0,2.0,3.0/,((B(I,J),J=1,2),I=1,3)/6*5./
```

Explanation

The *DATA* statement loads real numbers 1.0, 2.0, and 3.0 into array A. It also loads real number 5. into every element of array B. DATA statements must precede the first executable statement or statement function, and must follow any specification statements.

8.5 STATEMENT FUNCTIONS

A statement function is defined internal to the program unit in which it is referenced. All statement functions must precede the first executable statement and must follow any specification statements or DATA statements of the program unit.

The type declaration of a Statement Function may be accomplished in one of three ways: by the predefined convention, by the IMPLICIT statement, or by the explicit specification statements. Thus, the rules for declarating the type of variables apply to statement functions. The general form is

```
name(a1,a2,a3,...,an) = expression
```

where

name	is the statement function name
a	is a dummy argument. It must be a distinct variable (i.e., it may appear only once within the list of arguments). There must be at least one dummy argument.
expression	is any arithmetic or logical expression that does not contain array elements. Any statement function appearing in this expression must have been defined previously.

The expression to the right of the equal sign defines the operations to be performed when a reference to this function appears in a statement elsewhere in the program unit. The expression defining the function must not contain a reference to the function it is defining.

The dummy arguments enclosed in parentheses following the function name are dummy variables for which the arguments given in the function reference are substituted when the function reference is encountered. The same dummy arguments may be used in more than one statement function definition, and may be used as variables outside the statement function definitions. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

All statement function definitions to be used in a program must precede the first executable statement of the program.

Example

```
FUNC(A,B) = 3.*A+B**2.+X+Y+Z
```

Explanation

This example defines the Statement Function FUNC, where FUNC is the function name and A and B are the dummy arguments. The expression to the right of the equal sign defines the operations to be performed when the function reference appears in an arithmetic statement.

The function reference might appear in a statement as follows:

```
C = FUNC(D,E)
```

This is equivalent to:

```
C = 3.*D+E**2.+X+Y+Z
```

Note that correspondence between the dummy arguments A and B in the function definition and the actual arguments D and E in the function reference.

8.6 INTRINSIC FUNCTIONS

Intrinsic functions are commonly used subprograms contained in the FORTRAN library. The symbolic names and meanings of the intrinsic functions are shown in table 8-1.

An intrinsic function is referenced by a function call in an arithmetic expression. The arguments in the argument list must agree in type, number, and order with those shown in table 8-1.

Example

```
      IF (SIGN(W,X)) 1,2,2
1     W = ABS(X)-ABS(Y)
2     S = W*FLOAT(I*J)
      K = IFIX(X)+J
```

8.7 BASIC EXTERNAL FUNCTIONS

Basic external *FUNCTIONS* are standard subprograms contained in the FORTRAN library. These are referenced in the same manner as normal *FUNCTIONS*. The symbolic names and meanings of the basic external *FUNCTIONS* are shown in table 8-2.

8.8 DUMMY ARGUMENTS

Dummy arguments provide a means of passing information between a subprogram and the program or subprogram that called it. Both function and subroutine subprograms may have dummy arguments. A subroutine need not have any, while a function must have at least one. Dummies provide definitions of the data type, number, and sequence of subprogram parameters.

A dummy can be classified within a subprogram as a variable, an array, or an external procedure name. The actual arguments defined by a calling program or subprogram to which a dummy can correspond are: Hollerith constants, variables, array elements, arrays, expressions, and external procedure names.

Within a subprogram, a dummy can be used in much the same way as any other variable or array. A dummy can not appear in a *COMMON* or *EQUIVALENCE* statement.

The actual arguments (except for Hollerith constants) used in a calling statement agree in data type with the corresponding dummy arguments, that is, real to real, integer to integer, and array to array. If an actual argument is an expression, the result of the expression should correspond in data type to the dummy.

A dummy array is defined as an argument which appears in a *DIMENSION* statement in the subprogram. A dummy array does not occupy any storage but tells the subprogram that the argument supplied in the calling statement defines the first element of an actual array. The calling argument need not have the same dimensions as the dummy array. Useful operations can sometimes be performed by defining different dimensions for the dummy and calling arguments.

Example

```
DIMENSION       A(10,10)
CALL            FM(A(6,1))
  .
  .
  .
SUBROUTINE      FM(B)
DIMENSION       B(50)
```

For this case, one-dimensional dummy array B corresponds to the last half of two-dimensional array A. If the calling statement were CALL FM (A), dummy array B would correspond to the first half of array A.

8.9 ADJUSTABLE DIMENSIONS

As shown in the previous examples, the maximum value of each subscript in an array is specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if any array is used in a subprogram and is not in *COMMON*, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the specification statement, appearing in a subprogram, may contain integer variables that specify the size of the array. These integer variables must be either actual or implicit subprogram arguments. When the subprogram is called, these integer variables receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is called. Integer variables that provide dimension information may not be redefined within the subprogram.

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions of an array, appearing in a subprogram, should be less than or equal to the absolute dimensions of that array as declared in the calling program.

The following example illustrates the use of adjustable dimensions.

Example

```
      CALLING PROGRAM                    SUBPROGRAM
DIMENSION A(5,5)                  SUBROUTINE MAPMY (...,R,L,M,...)
      .                               .
      .                               .
      .                               .
CALL MAPMY(...,A,2,3,...)         DIMENSION...,R(L,M),...
      .                               .
      .                               .
      .                               .
                                  DO 100 I = 1,L
                                      .
                                      .
                                      .
```

Explanation

The statement DIMENSION A(5,5) appearing in the calling program declares the absolute dimensions of array A. When subroutine MAPMY is called, dummy argument R assumes array name A and dummy arguments L and M assume the values 2 and 3, respectively. The correspondence between the subscripted variables of arrays A and R is shown in the following example.

```
R(1,1)R(2,1)R(1,2)R(2,2)R(1,3)R(2,3)

A(1,1)A(2,1)A(3,1)A(4,1)A(5,1)A(1,2)A(2,2)...
```

Thus, in the calling program the subscripted variable A(1,2) refers to the sixth subscripted variable in array A. However, in subprogram MAPMY, the subscripted variable R(1,2) refer to the third subscripted variable in array A, namely, A(3,1). This is so because the dimensions of array R as declared in the subprogram are not the same as those in the calling program.

If the absolute dimensions in the calling program were the same as the adjusted dimensions in the subprogram, the subscripted variables R(1,1) through R(5,5) in the subprogram would always refer to the same storage locations as specified by the subscripted variables A(1,1) through A(5,5) in the calling program, respectively.

The numbers 2 and 3, which become the adjusted dimension of dummy array R, could also have been variables in the argument list or implicit arguments in a *COMMON* block. For example, assume that the following statement appeared in the calling program.

```
CALL MAPMY (...,A,I,J,...)
```

Then as long as the values of I and J are previously defined, the arguments may be variables. In addition, the variable dimension size may be passed through more than one subprogram level. For example, the subprogram MAPMY could have contained a call statement to another subprogram in which dimension information about A could have been passed.

Dummy variables (e.g., L and M) may be used as dimensions of an array only in a *FUNCTION* or *SUBROUTINE* subprogram.

8.10 COMBINING FORTRAN AND DAS MR

FORTRAN generates the following calling sequence for all implicit and explicit calls to subprograms:

```
JMPM      s
DATA      P1
DATA      P2
 .         .
 .         .
 .         .
DATA      Pn
```

where

s is the subprogram name

n is the number of arguments

P1, P2, and Pn are the addresses (not the value) of the arguments; these addresses can be direct or indirect.

If the above calling sequence is used, DAS MR programs can reference any program in the system library or any FORTRAN coded subprogram.

DAS MR subprograms to be used with FORTRAN must process the above calling sequence. The library program $SE can be used to transfer parameters by coding the DAS MR subprogram entry as follows:

```
s     ENTR
      CALL $SE
      DATA n
      BSS n
```

where

s	is the subprogram name
n	is the parameter count

$SE transfers the n parameter addresses, resolving indirect addresses sequentially into the block defined by BSS n. In addition, $SE increments the address in s so that the program returns to the address following the calling sequence.

The above calling sequence does not define a parameter count so it is difficult to use with subprograms that process a variable-length parameter list. The only library programs of this type are the intrinsic functions that list maximum and minimum values. The FORTRAN compiler detects calls to these values and outputs an absolute zero to mark the end of the parameter list. DAS MR programs can reference these functions by terminating the calling sequence with an absolute zero (not a pointer to zero).

Table 8-1. Intrinsic Functions

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	$a_1 \pmod{a_2}$	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing Largest Value	Max $(a_1, a_2, \ldots)$	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing Smallest Value	Min $(a_1, a_2, \ldots)$	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double

Table 8-1. Intrinsic Functions *(continued)*

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Positive Difference	a_i - min (a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double-Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single-Precision Argument in Double-Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	COMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	Conjg	Complex	Complex

* The function MOD or AMOD (a_1, a_2) is defined as a_i - [a_1 /a_2] a_2, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Table 8-2. Basic External Functions

External Functions	Definition	Arguments	Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	Real	Real
			DEXP	Double	Double
			CEXP	Complex	Complex
Natural Logarithm	$\log_e$ (a)	1	ALOG	Real	Real
			DLOG	Double	Double
			CLOG	Complex	Complex
Common Logarithm	$\log_{10}$ (a)	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	sin(a)	1	SIN	Real	Real
			DSIN	Double	Double
			CSIN	Complex	Complex
Trigonometric Cosine	cos(a)	1	COS	Real	Real
			DCOS	Double	Double
			CCOS	Complex	Complex
Hyperbolic Tangent	tanh(a)	1	TANH	Real	Real
Square Root	(a)	1	SQRT	Real	Real
			DSQRT	Double	Double
			CSQRT	Complex	Complex

Table 8-2. Basic External Functions *(continued)*

External Functions	Definition	Arguments	Name	Type of Argument	Type of Function
Arctangent	arctan(a)	1	ATAN	Real	Real
			DATAN	Double	Double
	arctan(a_1 /a_2)	2	ATAN2	Real	Real
			DATAN2	Double	Double
Remaindering*	a_1 (mod a)	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

* The function DMOD (a_1, a_2) is defined as $a_1 \cdot [a_1 / a_2]\, a_2$, where [x] is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

SECTION 9
VORTEX OPERATING PROCEDURES

This section contains operating procedures for FORTRAN IV programming systems that are used with VORTEX.

9.1 COMPILING WITH VORTEX

The initiation of the VORTEX FORTRAN IV compiler is accomplished by entering the control directive:

/FORT,P1,P2,...,Pn.

This control directive directs the executive program to call the system loader to load the FORTRAN IV compiler and commence compilation. The parameter string specifies optional tasks that are to be performed. These options are:

Parameter	Presence	Absence
A	Produce DASMR listing	Suppress DASMR listing
B	Suppresses binary object	Output binary object
D	Allocates two words to integer array items and to integer and logical variables (ANSI standard). This has no effect on range: 1-word integers still are in the range ±32767 (the second word is unused).	Allocates one word to integer array items and to integer and logical variables
H	Generate code using Floating-Point Processor (FPP)	Generate no FPP instructions
L	Outputs binary object on GO file	Suppresses output of binary object on GO file
M	Suppresses symbol-table listing	Outputs symbol-table listing
N	Suppresses source listing	Outputs source listing
O	Outputs object-module listing	Suppresses object-module listing
X	Compiles conditionally	Compiles normally
F	Generates code with calls to faster firmware routines	Generates subroutine calls

The /FORT directive can contain such parameters in any order.

Input/output assignments during compilation are made through the /ASSIGN and /PFILE control directives. The FORTRAN IV compiler uses the following logical units:

Source input	PI
Object output	BO
Listing	LO
Load and go	GO (optional)

9.2 LOAD AND GO OPERATION

FORTRAN programs may be compiled and executed on a LOAD and GO basis by setting the compiler operation switch 'L' ('B' is optional if permanent binary is not desired) and using the JCP directive /EXEC upon final compulation. Using this method the program may be reexecuted by successive /EXEC until SW is modified by another VORTEX program (i.e., LMGEN or another LOAD and GO operation). Note: LUN assignments must be made (if needed) prior to the /EXEC command.

9.2.1 Compiling and Cataloging Operation

The object program output by the VORTEX compiler is input to the load module generator (LMGEN). The job-control processor schedules LMGEN upon inputting the directive: /LMGEN. LMGEN creates a load module on the system-workfile SW device on inputting the following four directives:

TIDB,name, bf,s,DEBUG	bf = 1 for background: 2 for foreground s = Overlay count DEBUG is optional and loads the DEBUG routine when present
LD,obj	obj = specifier giving object module logical unit number and key, lun/key
LIB,lib	lib = specifier giving library lun/key
END,save	save is optional for specifying the load module save lun/key

The program can then be loaded and executed from SW by entering /EXEC on the System Input (SI) device; or, if bf = 1 and save = BL,E, it can be executed by the JCP directive /LOAD, name; or if bf = 2, it can be scheduled by entering the OPCOM directive 'SCHED,name,level,save, or by another task, using the SCHED macro.

9.2.2 Overlays

FORTRAN programs can be generated with or without overlays. The FORTRAN calling sequence for overlay is

CALL OVLAY (type, reload, name, parameters)

where

type	is 0 (default value) for load and execute, or 1 for load and return following the request.
reload	is a constant or name with the value zero to load or non-zero to load only if not currently loaded.
name	is a three-word Hollerith array containing the overlay segment name.
parameters	is optional, and, is the number of parameters that must correspond to the overlay subroutine formal parameter count.

FORTRAN overlays must be subroutines if called by a FORTRAN CALL. For example, find, load, and execute overlay segment OVSG01 without return.

```
DIMENSION N1(3)
DATA N1(1),N1(2),N1(3)/2HOV,2HSG,2H01/
CALL OVLAY(0,0,N1)
```

or

```
CALL OVLAY(0,0,6HOVSG01)
```

External subprograms may be referenced by overlays. If a subprogram S is called in several overlays, and S is not in the main segment, each overlay will be built with a separate copy of S.

Refer to the VORTEX reference manual sections 2.1.8 and 6.1.1, for more information on overlays.

9.2.3 Resident Programs

FORTRAN generated programs may be made resident under VORTEX II by using the SGEN TSK directive.

The object program output by the VORTEX compiler is input to the SGEN program and made part of the VORTEX nucleus. All required subroutines must be added at this time.

Note: Almost all FORTRAN programs require several OM library modules to be cataloged. Therefore, these OM modules must be included with the FORTRAN generated modules before the FORTRAN programs can be handled by VORTEX SYSGEN. Whenever possible, it is suggested that FORTRAN programs are not processed by SYSGEN.

9.3 I/O DEVICE CONTROL

The I/O control components of VORTEX permit access to I/O devices through the use of logical units. A logical unit is an I/O device or partition of a rotating-memory device (RMD). A program references an assigned number. The logical unit numbers permit I/O operations independent of the physical-device configuration. For further information on logical units, refer to the input/output control description in the VORTEX reference manual.

The FORTRAN IV compiler inputs source text from logical unit PI, outputs listings and maps on logical unit LO, and produces an object module (code and loading information) on logical units BO and GO. For further information, refer to the FORTRAN IV compiler description in the VORTEX reference manual.

9.4 COMPILER INPUT RECORDS WITH VORTEX

The compiler requests 40-word (80-character) input records from IOCS, if PI is not a rotating memory device (RMD) or if PI = SI. Otherwise, the compiler inputs 120-word records (three FORTRAN source records) from the RMD, and does its own deblocking. FORTRAN RMD source modules must start on a record boundary.

9.5 COMPILER OUTPUT RECORDS WITH VORTEX

Output records are 60 words long. An object module produced on an RMD is blocked two records for each RMD record. FORTRAN object modules start on the RMD-record boundary. The VORTEX reference manual describes the object module format.

9.6 ERROR MESSAGES

During compilation, source statements are checked for such items as validity, syntax, and usage. When an error is detected, it is posted on the LO usually beneath the source statement. The errors marked T terminate binary output.

All error messages are of the form

ERR xx c(1)-c(16)

where xx is a number from 0 to 18 (notification error), or T followed by a number from 0 to 9 (terminating error); and c(1)-c(16) is the last character string (up to 16) encountered in the statement being processed. The right-most character indicates the point of error and the @ indicates the end of the statement. The possible error messages are:

Notification Error	Definition
0	Illegal character input
1	Construction error
2	Usage error
3	Mode error
4	Illegal DO termination
5	Improper statement number
6	Common base lowered
7	Illegal equivalence group
8	Reference to nonexecutable statement
9	No path to this statement
10	Multiply defined statement number
11	Invalid format construction
12	Spelling error
13	Format statement with no statement number
14	Function not used as variable
15	Truncated value
16	Statement out of order
17	More than 29 named common regions
18	Noncommon data
19	Illegal name
20	DO index not referenced
21	Name is dummy
22	Array name previously declared
23	Exponent underflow or overflow
24	Undefined statement number

Terminating Error	Definition
T0	I/O error
T1	Construction error
T2	Usage error
T3	Data pool overflow
T4	Illegal statement
T5	Improper use
T6	Improper statement number
T7	Mode error
T8	Constant too large
T9	Improper DO nesting
T10	DO not parenthesized
T11	Item not operand
T12	Item not function
T13	Invalid unary +,
T14	Invalid hierarchy
T15	Invalid =
T16	Illegal operator
T17	Function statement without parameters
T18	Logical If follows logical If
T19	Invalid dimensions

Terminating Error	Definition
T20	Operand is not a name
T21	Too may numeric characters
T22	Non-numeric exponent
T23	Terminator not
T24	Illegal terminator
T25	Not statement end
T26	Invalid common type
T27	Target statement precedes DO
T28	Subscript variable not dummy
T29	Not first statement (Title statement)
T30	First two characters not DO
T31	Not in subprogram
T32	Subscript not integer constant

Note: due to optimization, the error message may appear on the next labeled statement and not on the actual statement error.

RUNTIME

When an error is detected during runtime execution of a program, a message is posted on the LO device of the form:

taskname message

Fatal errors cause the job to be aborted; execution continues for non-fatal errors. The messages and their definitions are:

Message	Cause
ARITH OVFL	Arithmetic overflow
GO TO RANGE	Computed GO TO out of range*
FUNC ARG	Invalid function argument (e.g., square root of negative number)
FORMAT	Error in FORMAT statement*
MODE	Mode error (e.g., outputting real array with I format)*
DATA	Invalid input data (e.g., inputting a real number from external medium with I format)*
I/O	I/O error (e.g., parity, EOF)*

* indicates fatal error; all others non-fatal

APPENDIX A
VORTEX FORTRAN IV LANGUAGE COMPARISONS

This appendix shows FORTRAN IV language feature comparisons. The SPERRY UNIVAC VORTEX FORTRAN language has all the capabilities of IBM level G FORTRAN for the 360/370 series computers except for:

- NAMELIST
- Call-by-name arguments
- Default integer size is *2 as opposed to *4
- PRINT, PUNCH, and READ with default unit identifier
- Double precision complex data type
- PROGRAM statement
- Right-to-left operation of multiple exponentiation
- DATA statement after the first executable statement
- DATA statement before specification statement
- Optional size for logical data type
- Sense switch/light

The VORTEX FORTRAN IV language conforms to the American National Standards Institute (ANSI) FORTRAN specification except for:

- The = character of replacement statements and statement functions must occur on the first line.
- The , character of DO statements must occur on the first line.
- Data statements must follow all specification statements and statement functions, and precede all executable statements and FORMAT statements.

FEATURES	ANSI x3.9	ANSI x3.10	IBM 360/370 Level G	SPERRY UNIVAC
Alternate RETURN	no	no	yes	yes
Adjustable dimension	yes	no	yes	yes
Array dimensions	3	2	7	7
ASSIGN, ASSIGNED GO TO	yes	no	yes	yes
AUXILIARY I/O				
REWIND	yes	yes	yes	yes
BACKSPACE	yes	yes	yes	yes
ENDFILE	yes	yes	yes	yes
BLOCK DATA	yes	no	yes	yes
Blanks in numeric conversions, non-leading	zero	no	zero	zero
Call-by-name	no	no	yes	no
CALL LINK	no	no	no	no
CALL LOAD	no	no	no	no
CHARACTER SET				
A-Z, 0-9	yes	yes	yes	yes
blank = + - * / ( ) ,	yes	yes	yes	yes
$	yes	no	yes	yes
'	no	no	yes	yes
COMMON				
Blank	yes	no	yes	yes
Named	yes	no	yes	yes
Dimensioned	yes	no	yes	yes
CONSTANTS				
complex	yes	no	yes	yes
logical	yes	no	yes	yes
literal	H	no	H,'	H,'
hexadecimal	no	no	yes	yes
octal	yes	yes	yes	yes

FEATURES	ANSI x3.9	ANSI x3.10	IBM 360/370 Level G	SPERRY UNIVAC
real				
basic real	yes	yes	yes	yes
integer with decimal exp.	yes	no	yes	yes
double precision real with "D" in place of "E"	yes	no	yes	yes
Continuation lines	19	5	19	19+
DATA	yes	no	yes	yes
Must be before first executable statement	no	-	no	yes
Must be after last specification statement	no	-	no	yes
Data types (default size, optional size)				
Integer	yes	yes	4,2	2,4
Real	yes	yes	4,8	4,8
Double precision	yes	no	8	8
Complex	yes	no	8,16	8
Logical	yes	no	4,1	2
Hollerith	yes	no	1	1
ENTRY	no	no	yes	yes
ERR, END I/O Specifiers	no	no	yes	yes
Extended Range DO	yes	no	yes	yes
EXTERNAL	yes	no	yes	yes
FORMAT				
Level of parentheses	2	1	3	3
Run-time	yes	no	yes	yes
Types A	yes	no	yes	yes
D	yes	no	yes	yes

FEATURES	ANSI x3.9	ANSI x3.10	IBM 360/370 Level G	SPERRY UNIVAC
E	yes	yes	yes	yes
F	yes	yes	yes	yes
G	yes	no	yes	yes
H	yes	yes	yes	yes
'	no	no	yes	yes
I	yes	yes	yes	yes
L	yes	no	yes	yes
P (scale factor)	yes	no	yes	yes
T	no	no	yes	yes
X	yes	yes	yes	yes
Z	no	no	yes	yes
Real Conversions				
E exponent	yes	yes	yes	yes
D exponent	yes	no	yes	yes
Generalized Subscript	no	no	yes	yes
Generic References	no	no	no	no
GLOBAL	no	no	no	no
IMPLICIT	no	no	yes	yes
INVOKE	no	no	no	no
I/O				
Unformatted	yes	yes	yes	yes
Formatted	yes	yes	yes	yes
1st character				
not printed	yes	yes	yes	yes
blank (space 1 line before)	yes	no	yes	yes
0 (space 2 lines before)	yes	no	yes	yes
1 (1st line, new page)	yes	no	yes	yes
+ (no advance)	yes	no	yes	yes
PRINT default unit	no	no	no	no
PUNCH default unit	no	no	yes	no

FEATURES	ANSI x3.9	ANSI x3.10	IBM 360/370 Level G	SPERRY UNIVAC
READ default unit	no	no	yes	no
Direct Access	no	no	yes	yes
DEFINE FILE	no	no	yes	yes
FIND	no	no	yes	yes
Logical IF	yes	no	yes	yes
Logical operators	yes	no	yes	yes
Mixed-mode expressions	no	no	yes	yes
Multiple exponentiation without parentheses	no	no	**	*
NAMELIST	no	no	yes	no
Numeric statement label	1-5	1-4	1-5	1-5
PAUSE				
digits	4	4	5	5
literal	no	no	yes	yes
PROGRAM statement	no	no	yes	no
Relational expressions	yes	no	yes	yes
Sense Switch/light Handling	no	no	yes	no
Specification Statements				
Precede 1st Executable	yes	yes	yes	yes
Ordered: DIMENSION, COMMON, EQUIVALENCE	no	yes	no	no
STOP				
digits	5	4	5	5
literal	no	no	yes	yes
Subprogram Arguments				
Hollerith	yes	no	yes	yes
Literal	no	no	yes	yes
External subprogram	yes	no	yes	yes
Define or redefine	yes	no	yes	yes

FEATURES	ANSI x3.9	ANSI x3.10	IBM 360/370 Level G	SPERRY UNIVAC
Type Statements				
INTEGER	yes	no	yes	yes
REAL	yes	no	yes	yes
COMPLEX	yes	no	yes	yes
LOGICAL	yes	no	yes	yes
DOUBLE PRECISION	yes	no	yes	yes
Dimensioned	yes	no	yes	yes
Data initialized	no	no	yes	yes
in FUNCTION statement	yes	no	yes	yes
Length specified	no	no	yes	yes
Variable Name size	1-6	1-5	1-6	1-6

* = left-to-right order
** = right-to-left order

APPENDIX B
V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
200	128	NUL			Null
201	129	SOH			Start of Heading
202	130	STX			Start of Text
203	131	ETX			End of Text
204	132	EOT			End of Transmission
205	133	ENQ			Enquiry
206	134	ACK			Acknowledge
207	135	BEL			Bell
210	136	BS			Backspace
211	137	HT			Horizontal Tab
212	138	LF			Line Feed
213	139	VT			Vertical Tab
214	140	FF			Form Feed
215	141	CR			Carriage Return
216	142	SO			Shift Out
217	143	SI			Shift In
220	144	DLE			Data Link Escape
221	145	DC1			Device Control 1
222	146	DC2			Device Control 2
223	147	DC3			Device Control 3
224	148	DC4			Device Control 4
225	149	NAK			Negative Acknowledge

V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
226	150	SYN			Synchronous File
227	151	ETB			End of Transmission Block
230	152	CAN			Cancel
231	153	EM			End of Medium
232	154	SUB			Substitute
233	155	ESC			Escape
234	156	FS			File Separator
235	157	GS			Group Separator
236	158	RS			Record Separator
237	159	US			Unit Separator
240	160	SP	(blank)	(blank)	Space
241	161	!	11/2/8	11/2/8	Exclamation Point
242	162	"	7/8	0/5/8	Quotation Mark
243	163	#	3/8	0/7/8	Pound Sign
244	164	$	11/3/8	11/3/8	Dollar Sign
245	165	%	0/4/8	11/7/8	Percent Sign
246	166	&	12	12/7/8	Ampersand
247	167	'	5/8	4/8	Apostrophe
250	168	(	12/5/8	0/4/8	Left Paren
251	169	)	11/5/8	12/4/8	Right Paren
252	170	*	11/4/8	11/4/8	Asterisk
253	171	+	12/6/8	12	Plus Sign
254	172	,	0/3/8	0/3/8	Comma
255	173	-	11	11	Minus Sign
256	174	.	12/3/8	12/3/8	Period

Octal	Decimal	Character	029	026	Description
257	175	/	0/1	0/1	Slash
260	176	0	0	0	
261	177	1	1	1	
262	178	2	2	2	
263	179	3	3	3	
264	180	4	4	4	
265	181	5	5	5	
266	182	6	6	6	
267	183	7	7	7	
270	184	8	8	8	
271	185	9	9	9	
272	186	:	2/8	5/8	Colon
273	187	;	11/6/8	11/6/8	Semi-Colon
274	188	<	12/4/8	12/6/8	Less Than
275	189	=	6/8	3/8	Equal Sign
276	190	>	0/6/8	6/8	Greater Than
277	191	?	0/7/8	12/2/8	Question Mark
300	192	@	4/8	0/2/8	At
301	193	A	12/1	12/1	
302	194	B	12/2	12/2	
303	195	C	12/3	12/3	
304	196	D	12/4	12/4	
305	197	E	12/5	12/5	
306	198	F	12/6	12/6	
307	199	G	12/7	12/7	

V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
310	200	H	12/8	12/8	
311	201	I	12/9	12/9	
312	202	J	11/1	11/1	
313	203	K	11/2	11/2	
314	204	L	11/3	11/3	
315	205	M	11/4	11/4	
316	206	N	11/5	11/5	
317	207	O	11/6	11/6	
320	208	P	11/7	11/7	
321	209	Q	11/8	11/8	
322	210	R	11/9	11/9	
323	211	S	0/2	0/2	
324	212	T	0/3	0/3	
325	213	U	0/4	0/4	
326	214	V	0/5	0/5	
327	215	W	0/6	0/6	
330	216	X	0/7	0/7	
331	217	Y	0/8	0/8	
332	218	Z	0/9	0/9	
333	219	[	12/2/8	12/5/8	Left Bracket
334	220	\	11/7/8	0/6/8	Backslash
335	221	]	0/2/8	11/5/8	Right Bracket
336	222	↑ or ^	12/7/8	7/8	Vertical Arrow
337	223	← or _	0/5/8	2/8	Horizontal Arrow

Octal	Decimal	Character	029	026	Description
340	224				Accent Grave
341	225	a			
342	226	b			
343	227	c			
344	228	d			
345	229	e			
346	230	f			
347	231	g			
350	232	h			
351	233	i			
352	234	j			
353	235	k			
354	236	l			
355	237	m			
356	238	n			
357	239	o			
360	240	p			
361	241	q			
362	242	r			
363	243	s			
364	244	t			
365	245	u			
366	246	v			
367	247	w			

V70 SERIES ASCII CHARACTER CODES

Octal	Decimal	Character	029	026	Description
370	248	x			
371	249	y			
372	250	z			
373	251	{			Left Brace
374	252	|			Vertical Line
375	253	}			Right Brace
376	254	~			Tilde
377	255	DEL			Delete, Rub Out

GLOSSARY

alphabetic character: a character of the set A,B,C,...,Z,$.

alphameric character: a character of the set which includes the alphabetic characters and the numeric characters.

argument: a parameter passed between a calling program and a subprogram or statement function.

arithmetic expression: a combination of arithmetic operators and arithmetic primaries.

arithmetic operator: one of the symbols +, -, *, /, **, used to denote, respectively, addition, subtraction, multiplication, division, and exponentiation.

arithmetic primary: an irreducible arithmetic unit; a single constant, variable, array element, function reference, or arithmetic expression enclosed in parentheses.

array: an ordered set of data items identified by a single name.

array declarator: the part of a statement which describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension. An array declarator may appear in a DIMENSION, COMMON, or type statement.

array element: a data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name: the name of an ordered set of data items.

assignment statement: an arithmetic or logical variable or array element, followed by an equal sign (=), followed by an arithmetic or logical expression.

basic real constant: a string of decimal digits containing a decimal point.

blank common: an unlabeled (unnamed) common block.

common block: a storage area that may be referred to by a calling program and one or more subprograms.

complex constant: an ordered pair of real constants separated by a comma and enclosed in parentheses. The first real constant represents the real part of the complex number; the second represents the imaginary part.

constant: a fixed and unvarying quantity. The four classes of constants specify numbers (numerical constants), truth values (logical constants), literal data (literal constants), and hexadecimal data (hexadecimal constants).

GLOSSARY

control statement: any of the several forms of GO TO, IF and DO statements, or the PAUSE, CONTINUE, and STOP statements, used to alter the normally sequential execution of FORTRAN statements, or to terminate the execution of the FORTRAN program.

data item: a constant, variable, or array element.

data type: the mathematical properties and internal representation of data and functions. The four basic types are integer, real, complex, and logical.

DO loop: repetitive execution of the same statement or statements by use of a DO statement.

DO variable: a variable, specified in a DO statement, which is initialized or incremented prior to each execution of the statement or statements within a DO loop. It is used to control the number of times the statements within the DO loop are executed.

dummy argument: a variable within a FUNCTION or SUBROUTINE statement, or statement function definition, with which actual arguments from the calling program or function reference are associated.

executable program: a program that can be used as a self-contained procedure. It consists of a main program and, optionally, one or more subprograms or non-FORTRAN-defined external procedures or both.

executable statement: a statement which specifies action to be taken by the program; e.g., causes calculations to be performed, conditions to be tested, flow of control to be altered.

extended range of a DO statement: those statements that are executed between the transfer out of the innermost DO of a completely nested nest of DO statements and the transfer back into the range of this innermost DO.

external function: a function whose definition is external to the program unit which refers to it.

external procedure: a procedure subprogram or a procedure defined by means other than FORTRAN statements.

formatted record: a record which is transmitted with the use of a FORMAT statement.

FUNCTION subprogram: an external function defined by FORTRAN statements and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

hexadecimal constant: the character Z followed by a hexadecimal number, formed from the set 0 through 9 and A through F.

hierarchy of operations: relative priority assigned to arithmetic or logical operations which must be performed.

implied DO: the use of an indexing specification similar to a DO statement (but without specifying the word DO and with a list of data elements, rather than a set of statements, as its range).

integer constant: a string of decimal digits containing no decimal point.

I/O list: a list of variables in an I/O statement, specifying the storage locations into which data is to be read or from which data is to be written.

labeled common: a named common block.

length specification: an indication, by the use of the form *s, of the number of bytes to be occupied by a variable or array element.

logical constant: a constant that specifies a truth value: true or false.

logical expression: a combination of logical primaries and logical operators.

logical operator: any of the set of three operators .NOT., .AND., .OR..

logical primary: an irreduceable logical unit: a logical constant, logical variable, logical array element, logical function reference, relational expression, or logical expression enclosed in parentheses, having the value true or false.

looping: repetitive execution of the same statement or statements, usually controlled by a DO statement.

main program: a program unit not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement and containing at least one executable statement. A main program is required for program execution.

name: a string of from one through six alphameric characters, the first of which must be alphabetic, used to identify a variable, an array, a function, subroutine, a common block, or a namelist.

nested DO: a DO loop whose range is entirely contained by the range of another DO loop.

nonexecutable statement: a statement which describes the use or extent of the program unit, the characteristics of the oprands, editing information, statement functions, or data arrangement.

numeric character: any one of the set of characters 0,1,2,...,9.

numeric constant: an integer, real, or complex constant.

predefined specification: the FORTRAN-defined type and length of a variable, based on the initial character of the variable name in the absence of any specification to the contrary. The characters I-N are typed INTEGER*4; the characters A-H, O-Z and $ are type REAL*4.

procedure subprogram: a FUNCTION or SUBROUTINE subprogram.

program unit: a main program or a subprogram.

range of a DO statement: those statements which physically follow a DO statement, up to an including the statement specified by the DO statement as being the last to be executed in the DO loop.

real constant: a string of decimal digits which must have either a decimal point or a decimal exponent, and may have both.

relational expression: an arithmetic expression, followed by a relational operator, followed by an arithmetic expression. The expression has the value true or false.

relational operator: any of the set of operators which express an arithmetic condition that can be either true or false. The operators are: .GT., .GE., .LT., .LE., .EQ., .NE., and are defined as greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to, respectively.

scale factor: a specification in a FORMAT statement whereby the location of the decimal point in a real number (and, if there is no exponent, the magnitude of the number) can be changed.

specification statement: one of the set of statements which provide the compiler with information about the data used in the source program. In addition, the statement supplies information required to allocate storage for this data.

specification subprogram: a subprogram headed by a BLOCK DATA statement and used to initialize variables in labeled (named) common blocks.

statement: the basic unit of a FORTRAN program, composed of a line or lines containing some combination of names, operators, constants, or words whose meaning is predefined to the FORTRAN compiler. Statements fall into two broard classes: executable and nonexecutable.

statement function: a function defined by a function definition within the program unit in which it is referenced.

statement function definition: a name, followed by a list of dummy arguments, followed by an equal sign (=), followed by an arithmetic or logical expression.

statement function reference: a reference in an arithmetic or logical expression to a previously defined statement function.

GLOSSARY

statement label: a number of from one through five decimal digits placed within columns 1 through 5 of the initial line of a statement. It is used to identify a statement uniquely, for the purpose of transferring control, defining a DO loop range, or referring to a FORMAT statement.

subprogram: a program unit headed by a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

SUBROUTINE subprogram: a subroutine consisting of FORTRAN statements, the first of which is a SUBROUTINE statement. It optionally returns one or more parameters to the calling program unit.

subscript: a subscript quantity or set of subscript quantities, enclosed in parentheses and used in conjunction with an array name to identify a particular array element.

subscript quantity: a component of a subscript: a positive integer constant, integer variable, or expression which evaluates to a positive integer constant. If there is more than one subscript quantity in a subscript, the quantities must be separated by commas.

type declaration: the explicit specification of the type and, optionally, length of a variable or function by use of an explicit specification statement.

unformatted record: a record for which no FORMAT statement exists, and which is transmitted with a one-to-one correspondence between internal storage locations (bytes) and external positions in the record.

variable: a data item that is not an array or array element, identified by a symbolic name.