IBM

Series/1

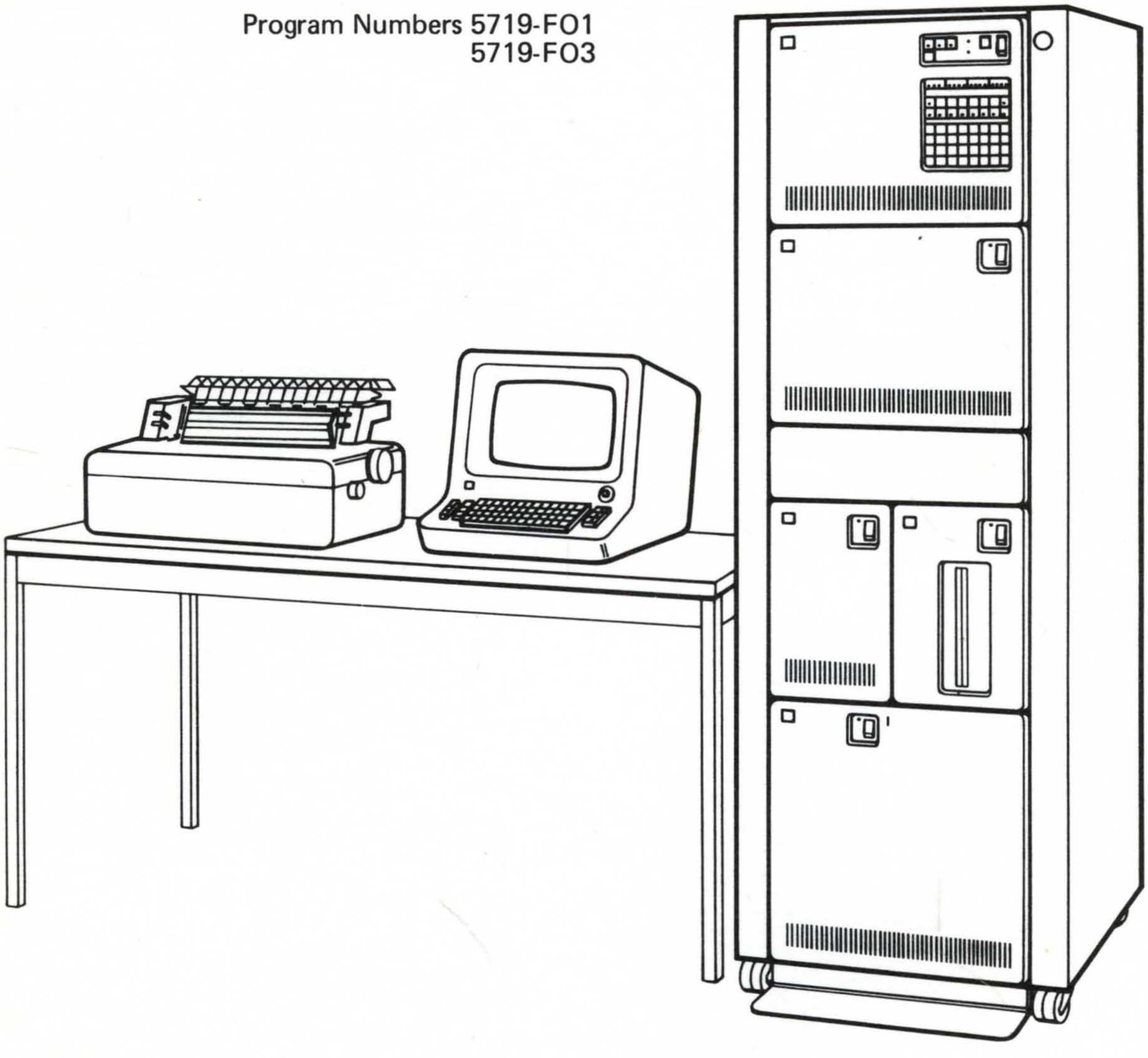
GC34-0133-0

S1-25

PROGRAM
PRODUCT

IBM Series/1
FORTRAN IV
Language Reference

Program Numbers 5719-FO1
5719-FO3



FORTRAN IV LANGUAGE REFERENCE

IBM

Series/1

GC34-0133-0

S1-25

PROGRAM
PRODUCT

FORTRAN IV LANGUAGE REFERENCE

IBM Series/1
FORTRAN IV
Language Reference

Program Numbers 5719-FO1
5719-FO3

This publication is for planning purposes only. The information herein is subject to change before the products described become available.

First Edition (February 1977)

This edition applies to IBM Series/1 FORTRAN IV (compiler and object support library), Program Number 5719-FO1, and IBM Series/1 Realtime Subroutine Library, Program Number 5719-FO3.

Significant changes or additions to the contents of this publication will be reported in subsequent revisions or Technical Newsletters. Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, send your comments to IBM Corporation, Systems Publications, Department 27T, P. O. Box 1328, Boca Raton, Florida 33432. Comments become the property of IBM.

Contents

Preface

This publication describes the Series/1 FORTRAN IV language. The language is a subset of American National Standard FORTRAN, X3.9-1966, and includes all of American National Standard (ANS) Basic FORTRAN, X3.10-1966, with the exception of object-time formats, adjustable dimensions, COMPLEX data type, G-format specifications, and two-level FORMAT parenthesis. Also included are IBM extensions to the language.

This publication, a language reference for the FORTRAN IV programmer who is developing realtime and batch applications for the Series/1 computer, presents the rules for coding FORTRAN IV statements and constructing the various kinds of program units. It is assumed that the reader is familiar with the basic coding techniques of FORTRAN programming.

Topics are presented in the following sequence:

1. General information about FORTRAN IV statements and how to code them.
2. Information pertaining to the values within a program: constants, variables, and arrays.
3. Descriptions and examples of:
   - Logical and arithmetic assignment statements and expressions
   - Control statements
   - Input/output statements
   - Data initialization statement
   - Specification statements.
4. Descriptions of FORTRAN IV subprograms and inter-program communication.

The appendixes include reference information regarding source program characters; FORTRAN IV-supplied mathematical functions, service subroutines, bit manipulator and interrogator functions, and address constant function; FORTRAN IV Realtime Subroutine Library, a program product providing realtime system support; non-standard integer lengths when using the NOCMPAT language compatibility option; a debug facility; sample programs; a comparison between Series/1 FORTRAN IV and other FORTRAN languages; and a glossary.

Associated Publications

IBM Series/1 FORTRAN IV: Introduction, GC34-0132
IBM Series/1 Realtime Programming System: Introduction and Planning Guide, GC34-0102
IBM Series/1 Program Preparation Subsystem: Introduction, GC34-0121
IBM Series/1 Mathematical and Functional Subroutine Library: Introduction, GC34-0138
IBM Series/1 FORTRAN IV: User's Guide (available July 1977)
IBM Series/1 Mathematical and Functional Subroutine Library: User's Guide (available July 1977)

Chapter 1. FORTRAN IV Statements

FORTRAN IV source programs consist of a set of statements from which the compiler generates execution-time instructions, constants, and storage areas. A given FORTRAN IV statement performs one of three functions:

- Causes certain operations to be performed
- Specifies the nature of the data being handled
- Specifies the characteristics of the source program

FORTRAN IV statements are composed of certain FORTRAN IV key words used with the elements of the language: constants, variables and expressions.

There are two broad classes of FORTRAN IV statements: executable and nonexecutable.

Executable statements may cause calculations to be performed, enable the user to transfer data between main storage and an input/output device, control the operation of those devices, change the order of execution of other statements in the program, or terminate program execution. An example of an executable statement is:

```
A = 96.0
```

This statement, an assignment statement, sets the variable named A to the value 96.0.

Nonexecutable statements may provide initial values for variables and array elements, specify the form in which data appears in FORTRAN IV records, define the properties of variables, arrays, and functions, declare the operations to be performed by statement functions, and name and specify arguments for subprograms. An example is:

```
DATA I/10/
```

This statement, a DATA statement, initializes the variable named I with a value of 10.

Some other examples of FORTRAN IV statements and their effects are:

`GO TO 10`	This statement says that the next statement to be executed is the one with the label 10.
`C=A/3`	The slash (/) indicates division. Thus, this statement means divide A by 3 and set C equal to the result. Using the data of the previous example, C would be given the value 32.

Coding FORTRAN IV Programs

Although the usual form of input to the computer is a sequential data set of card images, the initial coding of FORTRAN IV statements is generally on a coding sheet. The statements of a FORTRAN IV source program can be written on the standard FORTRAN IV coding form, GX28-7327.

The Series/1 FORTRAN IV compiler will accept an 80-column card image as input; however, the FORTRAN IV statement must be written only in columns 7–72 of each line of the coding form. Columns 1–5 may be used to write unsigned integer numbers by which the statement may subsequently be referenced. These statement numbers may be assigned in any order. Blanks and leading zeros in statement numbers are ignored by the FORTRAN IV compiler.

Thus, where ƀ denotes a blank,

```
00090
90ƀƀƀ
09ƀ0ƀ
ƀƀ90ƀ
```

are equivalent. However, a given statement number may appear in the label field only once in a program unit.

Columns 73–80 may be used for any desired identifying information. These columns are not analyzed by the compiler.

If a statement is too long for one line, it may be continued on as many as 19 successive lines by placing any character from the FORTRAN IV character set other than zero or blank in column 6. Column 1 of a continuation line may contain any character from the FORTRAN IV character set except the character 'C'. (The character 'C' is reserved for comments. See below for explanation.) Each of columns 2–5 of a continuation line may contain any character from the FORTRAN IV character set. In practice, columns 1–5 of a continuation line usually contain the blanks. However, the statement label of an initial line may be repeated in columns 1–5 of its respective continuation lines. These labels are not processed by the FORTRAN IV compiler, but are printed on its program listing. (If desired, the characters in column 6 may be used to indicate the order of continuation lines; that is, the character A may be inserted for the first continuation line, B for the second, etc.) Otherwise, column 6—for initial lines of a statement—must be blank or zero.

Blanks may be used to improve the readability of a FORTRAN IV program because the compiler ignores blanks except in certain limited cases (literal fields of statements and in column 6 of a card).

Thus,

```
A=B(I,J)-D-(C/E)-F**K
```

and

```
Aƀ=ƀBƀ(ƀIƀ,ƀJƀ)ƀ-ƀDƀ-ƀ(ƀCƀ/ƀEƀ)ƀ-ƀFƀ*ƀ*ƀK
```

are equivalent.

Comments to explain the program may be written in columns 2–72 of a line having a C in column 1. (The C in column 1 may be part of the comment.) Comments should not appear between continuation lines of a statement. Comments are not processed by the FORTRAN IV compiler, but are printed on its program listing.

Elements of the Language

In order to write FORTRAN IV programs, it is necessary to learn the rules for writing:

- Constants, such as 27 or 3.14159
- Variables, such as X or Y
- Array elements, such as X(I) or Y(3,2)
- Mathematical expressions, such as A+B or 3*J
- Assignment statements, which cause mathematical computations, such as a=b/c, which is written in FORTRAN IV as A=B/C
- Control statements, such as DO and GO TO, which affect the order in which statements are executed
- Specification statements, such as IMPLICIT, DATA, and COMMON, which provide the FORTRAN IV processor with information about the data used in the source program, and the amount of storage required for it

- Input/output statements, such as READ, WRITE, and FORMAT, which are used for getting data into the computer and for producing external results
- Subprogram statements, such as FUNCTION, which allow the programmer to cause specific processing to be performed without specifying each instruction every time the processing is to be done.

Order of a FORTRAN IV Program

The order of a FORTRAN IV program is as follows:

1. Subprogram statement for a subprogram. PROGRAM statement, if any, for a main program.
2. IMPLICIT statement, if any.
3. Other specification statements, if any.
4. Statement function definitions, if any, to describe statement functions.
5. Executable statements, at least one of which must be present.
6. END statement, to indicate the end of the program.

FORMAT and DATA statements (and ENTRY statements in a subprogram) may appear anywhere after IMPLICIT statement, if present, and before the END statement. DATA statements, however, must follow any specification statements that contain the same variable or array names. DEFINE FILE statements may appear anywhere after the IMPLICIT statement, but only in a main program. (See Appendix D for the order of debug statements.)

Chapter 2. Constants, Variables, and Arrays

Values within a program may take the form of constants, variables, or arrays.

Constants

A constant is a number which is used in computations without change from one execution of the program to the next. It appears in its actual numerical form in the source statement. For example, in the statement

```
J=3*X
```

3 is a constant, since it appears in actual numerical form.

Five types of constants may be specified in FORTRAN IV: integer constants (which are written without a decimal point or exponent), real constants (which are written with a decimal point or an exponent), logical constants, hexadecimal constants, and literal constants (which are strings of alphabetic, numeric, or special characters).

The rules for writing each of these constants are given in the following sections.

Integer Constants

An integer constant is a whole number written without a decimal point. A preceding + or − sign is optional. An unsigned constant is assumed to be positive.

All integer constants occupy four bytes of main storage. See Appendix C for a description of the NOCMPAT compile option. The magnitude of a four-byte integer constant (INTEGER*4) may not exceed 2147483647, or $2^{31}-1$.

Examples:

Valid integer constants:

```
0
+9
186
-327
6
45
123ƀ456
```

`123ƀ456`	(blank is ignored by compiler)

Invalid integer constants:

`4.321`	(contains a decimal point)
`-3,675`	(contains a comma)
`5436578656`	(exceeds the magnitude permitted)

Real Constants

- A basic real constant which is a number written with a decimal point, using the decimal digits 0, 1, ..., 9. A preceding + or − sign is optional. An unsigned constant is assumed to be positive.
- A basic real constant followed by a D or E, followed by a signed or unsigned one- or two-digit integer constant, which is the exponent. An unsigned exponent is assumed to be positive.
- An integer constant followed by a D or E, followed by a signed or unsigned one- or two-digit integer constant, which is the exponent.

In the exponent, the letter E specifies a single-precision constant occupying four bytes and the letter D specifies a double-precision constant occupying eight bytes. Unless it contains a D exponent, a real constant always occupies four bytes.

Magnitude:	Single-precision and double-precision constants have the same magnitude limitations: 0, or 16^{-65} (approximately 10^{-78}) through 16^{63} (approximately 10^{75}).
Precision:	Single-precision—6 hexadecimal digits, or approximately 7.2 decimal digits. Double-precision—14 hexadecimal digits, or approximately 16.8 decimal digits.

The decimal exponent permits the expression of a real constant as the product of a basic real constant or integer constant times 10 raised to a desired power.

Examples:

Valid real constants (single-precision) and equivalents

+0.	
-999.9999	
7.0E+0	($7.0 \times 10^0 = 7.0$)
19761.25E+1	($19761.25 \times 10^1 = 197612.5$)
7.E3	($7.0 \times 10^3 = 7000.0$)
7.0E3	($7.0 \times 10^3 = 7000.0$)
7.0E+03	($7.0 \times 10^3 = 7000.0$)
7E-03	($7.0 \times 10^{-3} = 0.007$)
21.4354657687	(*Note.* This level of precision cannot be accommodated in four bytes. FORTRAN IV truncates excess precision from the right.)

Valid real constants (double-precision) and equivalents

1234567890123456.D-93	($.1234567890123456 \times 10^{-77}$)
7.9D03	($7.9 \times 10^3 = 7900.0$)
7.9D+0	($7.9 \times 10^3 = 7900.0$)
7.9D+3	($7.9 \times 10^3 = 7900.0$)
7.9D0	($7.9 \times 10^0 = 7.9$)
7D03	($7.0 \times 10^3 = 7000.0$)

Invalid real constants

1	(Missing a decimal point or exponent)
3,471.1	(Embedded comma)
1.E	(Missing a decimal exponent following the E)
1.2E+113	(E is followed by a three-digit integer constant)
23.5E+97	(Magnitude outside the allowable range; 23.5×10^{97} greater than 16^{63})
21.3E-90	(Magnitude outside the allowable range; 21.3×10^{-90} less than 16^{-65})

Logical Constants

A constant that specifies a logical value true or false. There are two logical constants:

```
.TRUE.
.FALSE.
```

Each occupies four bytes of storage. The words TRUE and FALSE must be preceded and followed by periods.

Hexadecimal Constants

Hexadecimal constants are base 16 numbers that may only be used in the DATA initialization or explicit specification statements for specifying initial values for variables and array elements.

A hexadecimal constant consists of the character Z followed by a hexadecimal number formed from the set 0, 1, ..., 9, A, B, C, D, E, F.

In Series/1, one word contains four hexadecimal digits (two bytes). REAL, LOGICAL, and INTEGER*4 variables, therefore, would contain eight hexadecimal digits, and INTEGER*2 variables would contain four.

The internal form of each of the 16 possible digits is as follows:

0 - 0000	4 - 0100	8 - 1000	C - 1100
1 - 0001	5 - 0101	9 - 1001	D - 1101
2 - 0010	6 - 0110	A - 1010	E - 1110
3 - 0011	7 - 0111	B - 1011	F - 1111

If the number of digits is greater than the maximum, the leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied on the left.

Examples:

The eight-digit number Z1C49A2F1 represents the bit string

```
00011100010010011010001011110001
```

The seven-digit number ZBADFADE represents the bit string

```
00001011101011011111101011011110
```

where the first four zero bits are implied because an odd number of hexadecimal digits was written.

Further information about hexadecimal constants will be found in the section dealing with the DATA initialization statement.

Literal Constants

A literal constant is a string of alphabetic, numeric and/or special characters, delimited as follows:

- The string can be enclosed in apostrophes.
- The string can be preceded by wH where w is the number of characters in the string.

Each character requires one byte of storage. Note that the blank is considered a character. If apostrophes delimit the literal, a single apostrophe within the literal is represented by two apostrophes. If wH precedes the literal, a single apostrophe within the literal is represented by a single apostrophe.

Literals can be used in CALL statements or actual argument lists, as data initialization values, or in FORMAT statements. The first form, a string enclosed in apostrophes, may be used in PAUSE statements.

Examples:

```
'X-COORDINATE  Y-COORDINATE   Z-COORDINATE'
'3.14159'
'FRANCIS BACON''S ''HAMLET'''
```

Variables

A FORTRAN IV variable is a data item, identified by a name, that occupies a storage area. The value specified by the name is always the current value stored in the area.

For example, in the statement

```
A=5.0+B
```

both A and B are variables. The value of B has been determined by some previously executed statement. The value of A is calculated when the above statement is executed, and depends on the previously calculated value of B.

As with constants, a variable may be integer, real, or logical depending on whether the value it is to represent will be integer, real, or logical, respectively. Additionally, since a variable represents an area of storage, it is also assigned a length, either implicitly or explicitly. A real variable (REAL*4) has a length of four bytes. An integer variable can have a length of either two or four bytes (INTEGER*2 or INTEGER*4, respectively). A logical variable (LOGICAL*4) has a length of four bytes.

In order to distinguish between variables which will derive their value from an integer, as opposed to a real number, the rules for naming each type of variable are different, although these rules can be overridden.

Variable Names

A variable name consists of from 1 to 6 alphabetic or numeric characters, of which the first must be alphabetic. Blanks in a variable name are ignored.

Examples:

```
Ibbbbb
ABCDbb
BILL23
I$2bbb
ITEM1b
IbTbEbMb1
ITEMb1
```

In the above list, the last three names are considered to be identical.

The rules for naming variables allow for extensive selectivity. In general, it is easier to follow the flow of a program if meaningful names are used wherever possible. For instance, to compute distance it would be possible to use the statement:

```
A=B*C
```

but it might be more meaningful to write:

```
D=R*T
```

or

```
DIST = RATE * TIME
```

Variable Types

The type of a variable corresponds to the type of the data the variable represents. Variable type can be specified in three ways: predefined, implicitly, or explicitly.

Predefined Specification

If the first character of the variable name is I, J, K, L, M, or N, the variable is considered to be an INTEGER*[illegible] [illegible]. (See Appendix C for a description of the NOCMPAT compile option.) If the first character of the variable is not I, J, K, L, M, or N, the variable is REAL*4. There is no predefined specification for [illegible] variables.

Implicit Specification

By use of the IMPLICIT statement, the FORTRAN IV compiler can be [illegible] ignore the predefined convention for variable names [illegible] specified in the IMPLICIT statement. Thus, variables whose names begin with the letters I–N can be declared as [illegible] and [illegible] O–Z can be declared as type integer. In addition, [illegible] variables can be declared to be of the optional [illegible]. The [illegible] IMPLICIT statement will be given later.

Explicit Specification

Explicit specification [illegible] generally [illegible] variables by using the INTEGER, REAL, [illegible] statements. These statements override predefined [illegible]. The [illegible].

Arrays

An array is an ordered set of data items identified by a single name. A member of the array, called an array element, is identified according to its position by a quantity called a subscript.

Arrays, like variables and constants, have a type associated with them. The type of an array name is determined by the same conventions which govern the type of a variable name. Each element of an array is of the type and length specified for the array name.

Assume the following quantities are contained in an array named NEXT:

[illegible]

Suppose it is desired to refer to [illegible] mathematical notation this would be NEXT[illegible]. In FORTRAN IV this would be

[illegible]

The quantity 2 is the subscript.

Thus,

[illegible]

[illegible]

Similarly, ordinary mathematical notation might use NEXT[illegible] to represent any element of the set NEXT. In FORTRAN IV, this might be written as NEXT(I) where I equals 1, 2, 3, 4, or 5.

The array could be two dimensional, as for example the array [illegible]:

	Column 1	*Column 2*	*Column 3*
Row 1	82	4	7
Row 2	12	13	14
Row 3	91	1	31
Row 4	24	16	10
Row 5	2	8	2

Suppose it is desired to refer to the number in row 2, column 3; this would be

where JO(2,3)

2 and 3 are the subscripts. Thus,

JO(2,3) has the value 14
JO(4,1) has the value 24

Similarly, ordinary mathematical notation might use JOij to represent any element of the set JO. In FORTRAN IV, this might be written as JO(I,J) where I equals 1, 2, 3, 4, or 5, and J equals 1, 2, or 3.

The following rules apply to the construction of subscript quantities. (See the section "Arithmetic Expressions" for additional information.)

- Subscript quantities may consist of any arithmetic constant, arithmetic variable, arithmetic array element, or arithmetic expression.
- Mixed-mode expressions within subscript quantities are evaluated according to normal FORTRAN IV rules. If the evaluated expression is real, it is converted to integer.
- The evaluated result of a subscript quantity should always be greater than zero.
- In a subscript used in an I/O list, exponentiation and function references may not appear.

Examples:

Valid Array Elements:

```
ARRAY (IHOLD)
NEXT (19)
MATRIX (I-5)
BAK (I,J((K+1)*L,.3*A(M,N)))
ARRAY (I,J/4*K**2)
```

Invalid Array Elements:

ARRAY(1-5)	(A subscript quantity may not be negative)
LOT(0)	(A subscript quantity may not be nor assume a value of zero)
ALL(.TRUE.)	(A subscript quantity may not assume a true or false value)

Series/1 FORTRAN IV allows arrays of up to seven dimensions (seven subscript quantities).

The use of an array in the source program must be preceded by its declaration in either a DIMENSION statement, a COMMON or GLOBAL statement, or a type specification statement specifying the size of the array. These statements will be explained later.

Arrangement of Arrays in Storage

An array is stored in ascending storage locations, with the value of the first of its subscript quantities increasing most rapidly, and the value of the last increasing least rapidly.

For example, the array named A, described by one subscript quantity which varies from 1 to 5, appears in storage as follows:

```
A(1) A(2) A(3) A(4) A(5)
```

The array named B, described by two subscript quantities, with the first varying from 1 to 5 and the second from 1 to 3, appears in storage as follows:

```
B(1,1) B(2,1) B(3,1) B(4,1) B(5,1) B(1,2) B(2,2) B(3,2)
B(4,2) B(5,2) B(1,3) B(2,3) B(3,3) B(4,3) B(5,3)
```

Note that B(1,2) and B(1,3) follow in storage B(5,1) and B(5,2), respectively.

The following list is the order of a three-dimensional array, C(3,3,3):

```
C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1) C(2,2,1) C(3,2,1) C(1,3,1)
C(2,3,1) C(3,3,1) C(1,1,2) C(2,1,2) C(3,1,2) C(1,2,2) C(2,2,2)
C(3,2,2) C(1,3,2) C(2,3,2) C(3,3,2) C(1,1,3) C(2,1,3) C(3,1,3)
C(1,2,3) C(2,2,3) C(3,2,3) C(1,3,3) C(2,3,3) C(3,3,3)
```

Chapter 3. Assignment Statements and Expressions

Assignment statements and expressions are of two general types: arithmetic and logical.

Arithmetic Assignment Statements

The arithmetic assignment statement defines a numerical calculation; it very closely resembles a conventional arithmetic formula.

General Form of an Arithmetic Assignment Statement

```
a=b
```

where:

- a is a variable or array element
- b is an expression as defined below.

Examples:

The following are valid arithmetic assignment statements:

```
A=B+C
D( I )=E( I )+2.
```

In an arithmetic assignment statement, the equal sign means "is to be replaced" rather than "is equivalent to". An assignment statement is not an equation. This distinction is important. For example, suppose an integer variable I has the value 3. Then, the valid statement

```
I=I+1
```

would give I the value 4. This feature enables the programmer to keep counts and perform other required operations in the solution of a problem.

The following is an example of a series of arithmetic assignment statements:

`A=3.0`	Store the value 3.0 in A
`B=2.0`	Store the value 2.0 in B
`C=A+B`	Add the values in A and B and store in C (3.+2.=5.)
`C=C+1.`	Add 1. to the value C (5.+1.=6.)

Arithmetic Expressions

An expression in FORTRAN IV is a sequence of constants, variables, array elements, and operation symbols which indicate a quantity or a series of calculations. It must be formed according to the rules for constructing expressions. It may include parentheses and may also include functions (which will be discussed later). It may appear on the right-hand side of arithmetic assignment statements, in certain types of control and I/O statements, and as a subscript quantity.

Arithmetic Operation Symbols

Operation symbols used in FORTRAN IV are:

+	Addition
−	Subtraction

* Multiplication
/ Division
** Exponentiation

To express the arithmetic operation A = B divided by 1.7, the following statement may be used:

```
A=B/1.7
```

To express the operation J equals the product of K and L, this statement may be used:

```
J=K*L
```

Rules for Constructing Expressions

Since constants, variables, and array elements may be integer or real quantities, expressions may contain integer or real quantities; that is, two types may appear in the same expression. (In the following discussion, no mention is made of the rules for using integer and real quantities in functions. These rules will be stated when functions are discussed and will be considered as addenda to the following rules.)

1. The simplest expression consists of a single constant, variable or array element. If the quantity is an integer quantity, the expression is said to be of the integer type. If the quantity is a real quantity, the expression is said to be of the real type.

   Examples:

Expression	*Type of Quantity*	*Type of Expression*
3	Integer constant	Integer
3.0	Real constant	Real
I	Integer variable	Integer
A(J)	Real variable	Real
I(J)	Integer array element	Integer
A(J)	Real array element	Real

   In the last example, note that the subscript, which must be an integer quantity, does not affect the mode of the expression. The mode of the expression is determined solely by the type of the quantity itself.
2. Quantities may be preceded by plus or minus signs (+ or −), or may be connected by any of the operation symbols (+, −, *, /, **) to form expressions, provided:
   a. No two operation symbols appear consecutively. Quantities connected need not all be the same mode but will be converted to the higher mode (in the order I*2, I*4, R*4, R*8) before the expression is evaluated. For example, in A+I, if A is real and I is integer, I will be converted to real before the addition. Figure 3-1 shows the type and length of the result of arithmetic operations.
   b. No operation symbols are assumed to be present; that is, no two quantities appear consecutively.

   Valid expressions:

   -A+B
   B+C−J
   I/J
   K*L

   Invalid expressions:

A+−B	(must be written as A+(−B))
3J	(must be written as 3*J if multiplication is intended)

Parentheses may be used to specify the order of operations in an expression. In the absence of parentheses, operations with the same order of precedence are executed from left to right; successive exponentiations are evaluated from right to left.

Arithmetic Symbol	*Function*
**	Exponentiation
* and /	Multiplication and division
+ and −	Addition and subtraction

For example, the expression

A+B*C/D+E**F−G*H

will be taken to mean

$$A + \frac{B*C}{D} + E^{F} - (G*H)$$

Using parentheses, the expression

(A+B)*C/D+E**F−G*H

will be taken to mean

$$\frac{(A+B)*C}{D} + E^{F} - (G*H)$$

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=0−(B*C)
A=−B+C	is treated as	A=(0−B)+C

Second term / *First term*	*Integer (2)*	*Integer (4)*	*Real (4)*	*Real (8)*
Integer (2)	Integer (2)	Integer (4)	Real (4)	Real (8)
Integer (4)	Integer (4)	Integer (4)	Real (4)	Real (8)
Real (4)	Real (4)	Real (4)	Real (4)	Real (8)
Real (8)	Real (8)	Real (8)	Real (8)	Real (8)

Figure 3-1. Determining the type and length of the results of arithmetic operations

Types in an Arithmetic Assignment Statement

Expressions must be integer or real; however, the variable on the left-hand side of the equal sign in an arithmetic statement need not be of the same type as the expression on the right-hand side.

If the variable on the left is of type integer and the expression on the right is real, the expression will first be evaluated as a real quantity, the fractional portion will be dropped, and the remaining portion will be converted to an integer quantity. Thus, if the result is +3.872, the integer stored will be +3, not +4. If the variable on the left is real and the expression on the right is integer, the latter will be evaluated as an integer expression, and the result will be converted to real.

Examples:

Arithmetic Statement	*Result of Calculation*
A=3/2	A=1.
A=3./2	A=1.5
I=3/2	I=1
I=3./2.	I=1
I=3./2	I=1

Logical Assignment Statements and Expressions

The logical assignment statement is similar in form to the arithmetic assignment statement.

General Form of the Logical Assignment Statement

```
a=b
```

where:

- b is a logical expression
- a is either a logical variable or an element in a logical array.

Logical Expressions

The simplest form of logical expression is a single logical primary. A logical primary can be a logical constant, logical variable, logical array element, logical function reference, relational expression, or logical expression enclosed in parentheses. A logical primary, when evaluated, always has the value true or false.

Examples:

```
QTEST = .FALSE.
R( 17 )=B
```

where:

- QTEST and B are logical variables, and R is a logical array.

More complicated logical expressions may be formed by using logical operators to combine logical primaries.

Relational Expressions

Relational expressions are formed by combining two arithmetic expressions with a relational operator. The six relational operators, each of which must begin and end with a period, are as follows:

Relational Operator	*Definition*
.GT.	Greater than (>)
.GE.	Greater than or equal to (≥)
.LT.	Less than (<)
.LE.	Less than or equal to (≤)
.EQ.	Equal to (=)
.NE.	Not equal to (≠)

The relational operators express a relational condition existing between two arithmetic quantities which can be either true or false. The relational operators may be used to compare two integer expressions, two real expressions, or a real and an integer expression. The result of the relational expression is a logical value.

Examples:

Assume that the type of the following variables has been specified as follows:

Variable Names	*Type*
`ROOT, E`	Real variables
`A, I, F`	Integer variables
`L`	Logical variable

Then the following examples illustrate valid and invalid relational expressions.

Valid relational expressions:

```
E .LT.  I
E**2.7 .LE.  (5*ROOT+4)
.5 .GE.  .9*ROOT
E .EQ.   27.3E+05
```

Invalid relational expressions:

`L.EQ.(A+F)`	(Logical quantities cannot be joined by relational operators)
`E**2.LT97.1E1`	(Period missing immediately after the relational operator)
`.GT.9`	(Arithmetic expression missing before the relational operator)

Logical Operators

The three logical operators, each of which must begin and end with a period, are as follows (where A and B represent logical expressions):

Logical Operator	*Use*	*Meaning*
`.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.
`.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.

The only valid sequences of two logical operators are .AND..NOT. and .OR..NOT. (the sequence .NOT..NOT. is invalid unless used as unary operators).

Only those expressions which, when evaluated, have the value true or false may be combined with the logical operators to form logical expressions.

Examples:

Assume that the type of the following variables has been specified as follows:

Variable Names	*Type*
`ROOT, E`	Real variables
`A, I, F`	Integer variables
`L, W`	Logical variables

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

Valid logical expressions:

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

Invalid logical expressions:

`A.AND.L`	(A 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.
`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)

Order of Computations in Logical Expressions

The order in which the operations are performed is:

Operation	*Hierarchy*
Evaluation of functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and −); unary plus and minus	4th
Relationals (.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.)	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	

Note. Logical expressions may not require that all parts be evaluated. Functions within logical expressions may or may not be called. For example, in the expression A.OR.LGF(.TRUE.), it should not be assumed that the LGF function is always invoked, since it is not necessary to do so to evaluate the expression when A has the value true.

Use of Parentheses in Logical Expressions:
Parentheses may be used in logical expressions to specify the order in which operations are to be performed. The innermost pair of parentheses is evaluated first. For example, the logical expression:

```
.NOT.((B.GT.C.OR.K).AND.L)
```

is evaluated in the following order:

1.	B.GT.C	Call the result X	.NOT.((X.OR.K).AND.L)
2.	X.OR.K	Call the result Y	.NOT.(Y.AND.L)
3.	Y.AND.L	Call the result Z	.NOT.Z
4.	.NOT.Z	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.) is the equivalent of .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 is the equivalent of .TRUE.. Therefore, the value of the second expression is true. Note that the value of B is irrelevant to the result in this example. Thus, if B were a function reference, it would not have to be evaluated.

Chapter 4. Control Statements

Normally, FORTRAN IV statements are executed sequentially. However, it is often undesirable to proceed with each statement in this manner. This chapter will discuss some of the statements used to alter sequential execution, and why this may be desirable.

Unconditional GO TO Statement

This statement is used to interrupt sequential execution; it indicates the statement that is to be executed next.

General Form of the Unconditional GO TO Statement

```
GO TO n
```

where:

- n is the statement number of an executable statement.

This statement causes the statement whose number is n to be executed next.

Examples:

```
GO TO 16
GO TO 137
```

A coding example is shown below:

```
      •
      •
      •
      A=3.
      B=4.
      GO TO 7
12    B=2.*A
 7    A=2.*B
      •
      •
      •
```

Statement 12 will not be executed. After the GO TO statement is executed, statement 7 will be evaluated and A will be assigned the value 8.0.

Any executable statement immediately following the unconditional GO TO statement should have a statement number; otherwise, it can never be referred to or executed.

Computed GO TO Statement

This statement also indicates the statement that is to be executed next. However, it allows that statement to be different at various stages in the program.

General Form of the Computed GO TO Statement

GO TO $(n_1, n_2, \ldots, n_m)$, i

where:

- n_1, n_2,...,n_m are statement numbers of executable statements and i is an integer variable, not an array element, normally having a value between 1 and m, inclusive.

The parentheses enclosing the statement numbers, the commas separating the statement numbers, and the comma following the right parenthesis are all required punctuation.

This statement causes transfer of control to the 1st, 2nd, 3rd, etc., statement in the list depending on whether the value of i is 1, 2, 3,..., etc. If i has a value less than 1 or greater than the number of items in the list, the statement following the GO TO statement is executed next. The value that i has at any given time must be set by a preceding statement.

Examples:

`GO TO(5,7,8,2,4),J`	If J is 3, transfer control to statement 8.
`GO TO(4,4,4,7,8,9),MAX`	This example illustrates the fact that several values of i may cause a transfer of control to the same statement. In this case, when MAX has the values 1, 2, or 3, transfer of control will be made to statement number 4.

Further use of the computed GO TO is illustrated below:

```
      •
      •
      •
      A=3.
      B=4.
      C=5.
      K=0
1     K=K+1
      GO TO (10,20,30),K
      •
      •
      •
30    F=A-B
      GO TO 12
20    E=A-C
      GO TO 1
10    D=B-C
      GO TO 1
      •
      •
      •
12    CONTINUE
```

As a study of this example will show, D, E and F are computed, in that order, and control proceeds to statement 12. Of course, the example itself is highly simplified; if these were the only required calculations in this series, the programmer would just compute D, E, and F sequentially, in any desired order and without using the computed GO TO.

ASSIGN and Assigned GO TO Statements

General Form of the ASSIGN and Assigned GO TO Statements

ASSIGN i TO k
•
•
•
GO TO k, $(n_1,n_2,\ldots n_m)$

where:

- i is the number of an executable statement. It must be one of the numbers n_1, n_2, n_3,...,n_m.

- Each n is the number of an executable statement in the program unit containing the GO TO statement.
- k is an integer variable (not an array element) of length 4 which is assigned one of the statement numbers: n_1, n_2, n_3,...,n_m. See Appendix C for a description of the NOCMPAT compile option.

The assigned GO TO statement causes control to be transferred to the statement numbered n_1, n_2, n_3,..., or n_m, depending on whether the current assignment of k is n_1, n_2, n_3,..., or n_m, respectively. For example, in the statement:

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

If the current assignment of the integer variable n is statement number 8, then the statement numbered 8 is executed next. If the current assignment of n is statement number 10, the statement numbered 10 is executed next. If n is assigned statement number 25, statement 25 is executed next.

At the time of execution of an assigned GO TO statement, the current value of k must have been defined to be one of the values n_1, n_2,...n_m by the previous execution of an ASSIGN statement. Note that ASSIGN 10 TO I is not the same as I = 10.

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

Example 1:

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

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
    •
    •
```

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.

Logical IF Statement

The logical IF statement permits the programmer to execute or skip an associated statement depending on the value—true or false—of a relational expression.

General Form of the Logical IF Statement

```
IF (a) s
```

where:

- a is a logical expression and s is any executable statement except a DO statement or another logical IF statement.

Examples:

```
IF(A .GT.  1.0) GO TO 50
IF(A-B .LT.  A+C) A=B
IF (A .GT. B .OR. C .LT. D) GO TO 10
```

The associated statement is executed if the logical expression is true. Otherwise the statement following the IF statement is executed next. In the second example, if the logical expression is true, A is set equal to B and then the statement following the IF statement is executed.

Suppose a series of records, each containing a variable code number, I, is being read and processed. Certain of the records, appearing at random but with special code numbers greater than 99, are to be processed differently. The FORTRAN IV statements to accomplish this might be as follows:

```
     •
     •
     •
     IF(I.GT. 99) GO TO 20
     •
     •
     •
20   A=B+C
     •
     •
     •
```

Arithmetic IF Statement

This statement permits a programmer to change the sequence of statement execution, depending upon the value of an arithmetic expression.

General Form of the Arithmetic IF Statement

IF (a) n_1, n_2, n_3

where:

- a is an arithmetic expression and n_1, n_2, and n_3 are the statement numbers of executable statements.

  The expression, a, must be enclosed in parentheses; the statement numbers must be separated from one another by commas. The same statement number may be specified more than once.

Examples:

```
IF(A-B)10,10,7
IF(A(I)/D)1,2,3
```

Control is transferred to statement number n_1, n_2, or n_3 depending on whether the value of a is less than, equal to, or greater than zero, respectively. Note that in the first example, the same statement, numbered 10, is to be executed if A−B is less than or equal to 0.

As another example, suppose a value, A, is being computed. Whenever this value is positive, it is desired to proceed with the program. Whenever the value of A is negative, an alternative route starting at statement 12 is to be followed, and if A is zero, an error routine at statement 72 is to be executed. This may be coded as:

```
     •
     •
     •
     A=(B+C)/(D**E)-F
     IF(A)12,72,10
10   •
     •
     •
12   •
     •
     •
72   •
```

DO Statement

General Forms of the DO Statement

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

where:

- n is a statement number, i is an integer variable not an array element, and m_1, m_2, and m_3 are each either an unsigned integer constant or integer variable, not an array element, whose values are greater than 0. If m_3 is not stated, it is taken to be 1. The commas separating the parameters are required.

Examples:

```
DO 20 JBNO = 1,10
DO 20 JBNO = 1, 10, 2
DO 20 JBNO = K, L, 3
```

The DO statement is a command to execute repeatedly the statements which follow, up to and including the statement having statement number n. The first time, the statements are executed with $i=m_1$. For each succeeding execution of the statements, i is increased by m_3. After the statements have been executed with i equal to the highest value of this sequence which does not exceed m_2, control passes to the statement following the last statement in the range of the DO (the statement after statement n). Upon completion of the DO, the DO variable is undefined and may not be used until assigned a value (e.g., in an arithmetic assignment statement).

Looping and the DO Statement

The ability of a computer to repeat the same operations with different data, called looping, is a powerful tool which greatly reduces programming effort. There are several ways to accomplish this looping; one way is to use an IF statement. For example, assume that a plant carries 1,000 parts in inventory. Periodically it is necessary to compute stock on hand of each item (INV), by subtracting stock withdrawals of that item (IOUT) from previous stock on hand.

It would be wasteful to write a program which would indicate each separate subtraction by a separate statement. The same results could be achieved by the following statements:

```
     •
     •
     •
 5   J=0
10   J=J+1
25   INV(J)=INV(J)-IOUT(J)
15   IF(J .LT. 1000) GO TO 10
20   •
     •
     •
```

An index, J, is established which will be increased by 1 each time statement 10 is executed. Statement 5 initializes J to zero so that statement 10 will set J equal to 1 for the first execution of statement 25.

Statement 25 will compute the current stock on hand by subtracting the stock withdrawal from the previous stock on hand. The first time statement 25 is executed, the stock on hand of the first item in inventory, INV(1), will be computed by subtracting the stock withdrawal of that item, IOUT(1). Statement 15 tests whether all items in stock have been updated. If not, J will be less than 1000 and the program will transfer to statement 10, which will increment J by 1. Statement 25 will be executed again, this time for the stock on hand of item 2, INV(2), and the stock withdrawal of item 2, IOUT(2). This procedure will be repeated until the stock of item 1000 has been updated. At this point, J will not be less than 1000, causing execution to continue with statement 20.

Notice that three statements (5, 10 and 15) were required for this looping; this could have been accomplished with a single DO statement.

Not only does the DO simplify the programming of loops, it also provides greater flexibility in looping. Thus, the DO statement:

- Establishes an index which may be used as a subscript or in computations
- Causes looping through any desired series of statements, as many times as required
- Increases the index (by any positive amount that the programmer specifies) for each separate execution of the series of statements in the loop.

Example:

```
     •
     •
     •
15   DO 25 J=1,1000
25   INV(J)=INV(J)-IOUT(J)
35   •
     •
     •
```

Statement 15 is a command to execute the following statements up to and including statement 25. The first time J will be 1; thereafter J will be increased by 1 for each execution of the loop until the loop has been executed with J equal to 1000. After the loop has been executed with J equal to 1000, the statement following statement 25 will be executed.

The following is a comparison of statement 15 with the general form of the DO, and an introduction of some of the terms used in discussing DO statements:

$$\text{DO} \quad \underbrace{n}_{\text{Range}} \quad \underbrace{i}_{\text{Index}} = \underbrace{m_1,}_{\text{Initial value}} \quad \underbrace{m_2,}_{\text{Test value}} \quad \underbrace{,\ m_3}_{\text{Increment}}$$

$$\text{DO} \quad \underbrace{25}_{\text{Range}} \quad \underbrace{J}_{\text{Index}} = \underbrace{1,}_{\text{Initial value}} \quad \underbrace{1000}_{\text{Test value}}$$

The range is the series of statements to be executed repeatedly. It consists of all statements following the DO, up to and including statement n. In this case, statement n is statement 25, and the range consists of only one statement. The range can consist of any number of statements.

The index is the integer variable whose value will change for each execution of the range. In the example, this index was used as a subscript; in another problem it might be used in computations, etc. (The index need not be used in the range, although it usually is.)

The initial value is the value of the index for the first execution of the range. Although the initial value was 1 for this example, in another problem it might be some different integer quantity. Often, the initial value will change at different times within the program. In such cases it may be stated as an integer variable. The variable must then be assigned a value before the DO is executed.

The test value is the value which the index may not exceed. After the range has been executed with the highest value of the index which does not exceed the test value, the DO is satisfied, and the program continues with the first statement following the range. In the example, the DO was satisfied after the range was executed with the index equal to the test value. In some cases, the DO is satisfied before the test value is reached. Consider, for example, the following DO:

```
      DO 5 K=1, 9, 3
      •
      •
5     •
```

In this example, the range will be executed with K equal to 1, 4 and 7. The next value of K would be 10; since this exceeds the test value, control passes to the statement following statement 5 after the range is executed with K equal to 7. The test value may also be written as an integer variable.

The increment is the amount by which the value of the index will be increased after each execution of the range. In the example, this is not coded because the increment desired is 1, and the general form permits omission of the increment when it is 1. As with the initial value, the increment may be written as an integer variable.

As a further example, consider the following program:

```
      •
      •
      •
      K=0
      L=10
      DO 5 JOB=1, L, 2
      K=K+1
5     M(JOB=N(JOB)-K*JOB
```

This would cause the following computations:

M(1)=N(1)-1*1
M(3)=N(3)-2*3
M(5)=N(5)-3*5
M(7)=N(7)-4*7
M(9)=N(9)-5*9

When using DO statements, the following rules must be followed:

1. The indexing parameters of a DO statement (i, m_1, m_2, m_3) should not be changed by a statement within the range of the DO loop.
2. There may be other DO statements within the range of a DO statement. All statements in the range of an inner DO must be in the range of each outer DO. A set of DO statements satisfying this rule is called a nest of DO's.

Example 1:

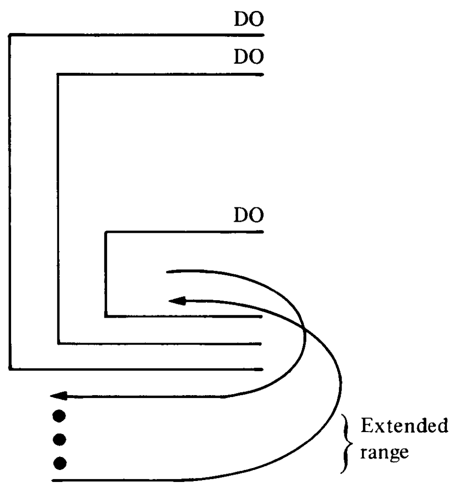
```
   DO 50 I = 1, 4
   A(I) = B(I)**2                    Range of
   DO 50 J=1, 5                      outer DO
50 C(I,J) = A(I)     Range of
                     inner DO
```

Example 2:

```
   DO 10 I = L, M
   N = I + K
   DO 15 J = 1, 100, 2                              Range of
15 TABLE(J, I) = SUM(J,N)-1      Range of           outer DO
10 B(N) = A(N)                   inner DO
```

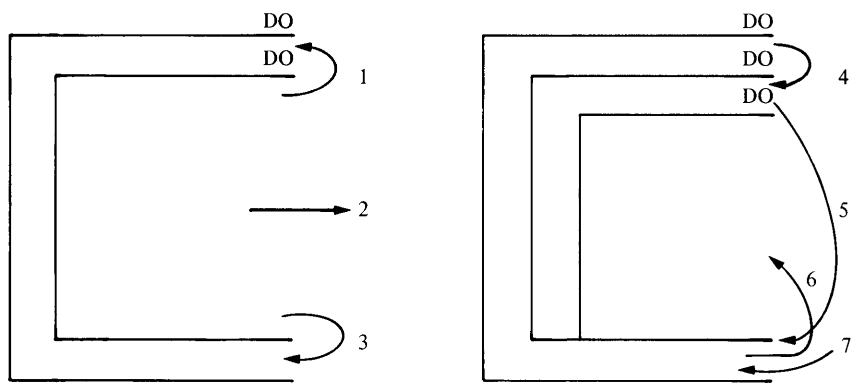
3. A transfer out of the range of any DO loop is permissible at any time. The DO variable is defined when such a transfer is executed, and its value is the value it has when the transfer is executed.
4. The extended range of a DO is defined as those statements that are executed between the transfer out of the innermost DO of a set of completely nested DO's and the transfer back into the range of this innermost DO. In a set of completely nested DO's, the first DO is not in the range of any other DO, and each succeeding DO is in the range of every DO which precedes it. The following restrictions apply:
   - Transfer into the range of a DO is permitted only if such a transfer is from the extended range of the DO.
   - The extended range of a DO statement must not contain another DO statement that has an extended range if the second DO is within the same program unit as the first.
   - The indexing parameters (i, m_1, m_2, m_3) cannot be changed in the extended range of the DO.

Example 3:



5. A statement that is the end of the range of more than one DO statement is within the innermost DO. The statement label of such a terminal statement may not be used as the branch target of any statement except those within the range of the innermost DO with that terminal statement.

Example 4:



In the preceding example, the transfers specified by the numbers 1, 2, and 3 are permissible, whereas those specified by 4, 5, 6, and 7 are not.

6. The indexing parameters (i, m_1, m_2, m_3) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement that uses those parameters.
7. The last statement in the range of a DO loop (statement x) must be an executable statement. It cannot be a GO TO statement of any form, or a PAUSE, STOP, RETURN, arithmetic IF statement, another DO statement, or a logical IF statement containing any of these forms.
8. The use of a subprogram within any DO loop (that is either in a nest of DO's or an extended range) is permitted.

CONTINUE Statement

This statement is primarily used as the last statement in the range of a DO where the last statement would otherwise violate DO rules.

General Form of a CONTINUE Statement

```
CONTINUE
```

As an example of a program which requires a CONTINUE, consider the following:

```
   •
   •
   •
10 DO 12 I=1, 100
   IF (ARG.EQ.VALUE(I)) GO TO 20
12 CONTINUE
   •
   •
   •
```

This program will scan the 100-element VALUE array until it finds an entry which equals the value of the variable ARG, whereupon it will transfer control to statement 20 with the value of I available for use. If no entry in the array equals the value of ARG, a normal exit to the statement following the CONTINUE will occur.

Note that a CONTINUE statement (when used as above) is meaningless without a statement number.

A CONTINUE statement may appear anywhere in the source program (where an executable statement may appear) without affecting the sequence of execution.

PAUSE Statement

This statement will cause a halt of the execution of the program and the display of a message. PAUSE, PAUSE n, or PAUSE 'message' is displayed, depending on how the source statement was coded.

General Forms of the PAUSE Statement

```
PAUSE
PAUSE n
PAUSE 'message'
```

where:

- n is an unsigned integer constant not greater than 99999, and 'message' is a literal constant, enclosed in apostrophes, containing up to 56 alphmeric and/or special characters.

The program issuing the PAUSE remains in a wait state until the operator responds by either striking the return key or typing any single character. (The character typed is not transmitted as data to the program.) The program then resumes executing at the next FORTRAN IV statement following the PAUSE.

STOP Statement

This statement terminates the execution of the program, and displays a halt code if an integer constant is specified (such as described under "PAUSE Statement" above).

General Forms of the STOP Statement

```
STOP
STOP n
```

where:

- n is an unsigned integer constant not greater than 99999.

If n is specified, the constant will be displayed.

END Statement

General Form of the END Statement

```
END
```

The END statement is a nonexecutable statement that defines the end of a main program or subprogram. Physically, it must be the last statement of each program unit. The END statement may not have a statement number and it may not be continued. The END statement does not terminate program execution. To terminate execution a STOP or RETURN statement in the main program is required.

Chapter 5. Input/Output Statements

Input/output statements are used to transfer and control the flow of data between internal storage and an input/output device, such as a programmer console or a disk storage unit.

Series/1 FORTRAN IV provides two types of input/output statements: sequential and direct access. Sequential I/O statements read or write records consecutively. Direct-access I/O statements read or write records in any chosen order.

Certain input/output devices are sequential. Only sequential I/O statements can be used in conjunction with the keyboard, paper tape unit, or a printer. However, on a disk storage unit, records can also be read or written directly; that is, in an order determined by the programmer. When records are to be read or written directly, direct-access I/O statements must be used. Note, however, that unformatted sequential I/O statements can be used to read or write records sequentially on the disk and that direct-access I/O statements can also be used on disk for reading or writing records sequentially.

A data set reference number in each input/output statement specifies which input/output device or data set on a device is to be used in the operation. Data set reference numbers identify data set definitions.

Input/output statements in FORTRAN IV are primarily concerned with the transfer of data between storage locations defined in a FORTRAN IV program and records which are external to the program. On input, data is taken from a record and placed into storage locations that are not necessarily contiguous. On output, data is gathered from diverse storage locations and placed into a record. An I/O list is used to specify which storage locations are used. The I/O list can contain the names of variables, array elements, or arrays, or a form called an implied DO.

Sequential Input/Output Statements

Sequential READ and WRITE statements may process formatted records, unformatted records, or list-directed records.

A formatted record has a FORMAT statement associated with it. A FORMAT statement specifies the form of data on the external medium. (The FORMAT statement will be explained in greater detail later in this section after the I/O statements themselves have been described.) Any number of records may be read or written with one execution of a formatted READ or WRITE statement. The data in the records are converted according to specifications listed in the FORMAT statement and are assigned to, or taken from, elements listed in the READ or WRITE statement, respectively.

An unformatted record has no FORMAT statement associated with it. Only one record may be transmitted per execution of an unformatted READ or WRITE statement. The unformatted READ is generally used to read records which have been written on a disk by an unformatted WRITE statement.

A list-directed record is similar to an unformatted record in that it has no FORMAT statement associated with it. However, it is used to transmit records to and from unit-record devices such as a keyboard.

The READ Statement

General Forms of the READ Statement

```
READ (u,f,END=s,ERR=t) list
READ (u,END=s,ERR=t) list
READ (u,*,END=s,ERR=t) list
```

where:

- u is an unsigned integer constant or integer variable of length 4 which is the data set reference number of the device to be read from. See Appendix C for a description of the NOCMPAT compile option.
- * specifies list-directed data mode for unit-record devices without use of a FORMAT statement.
- f is the statement number of the FORMAT statement describing the data items to be read.
- END=s is optional and specifies the number (s) of an executable statement to which control is to be transferred if end-of-file is encountered. Statement s must be in the same program unit as the READ statement.
- ERR=t is optional and specifies the number (t) of an executable statement to which control is to be transferred if a transmission error occurs during the data transfer. Statement t must be in the same program unit as the READ statement.
- If END or ERR is not specified, the preceding comma is omitted.
- list is an I/O list and is optional.

Although the END and ERR parameters need not be specified, if an end-of-file condition is encountered or a transmission error occurs and the appropriate parameter is not present, execution of the program may terminate. (See the discussion of the service subprogram ERRXIT in Appendix B.) END and ERR may appear in any order within the parentheses, but must follow the data set reference number and FORMAT statement number, if present.

Examples:

```
READ(9,100) D,E,F
```

This formatted READ statement causes data to be read from the data set whose reference number is 9 into the variables D, E, and F, in the format specified by the FORMAT statement numbered 100.

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

This unformatted READ statement causes data to be read from the data set whose reference number is the current value of J into the variables A, B, and C.

```
READ (1,*,END=200) (ARRAY(I),I=1,25),B(1),C(6)
```

This list-directed READ statement causes data to be read from the data set whose reference number is 1 into the 27 array elements specified by the list. If an end-of-file record is encountered, control is transferred to the statement numbered 200.

The WRITE Statement

General Forms of the WRITE Statement

```
WRITE (u,f,ERR=t) list
WRITE (u,ERR=t) list
WRITE (u,*) list
```

where:

- u is an unsigned integer constant or integer variable of length 4 which is the data set reference number of the device to be written to. See Appendix C for a description of the NOCMPAT compile option.

- * specifies list-directed data mode for unit record devices without use of a FORMAT statement.
- f is the statement number of the FORMAT statement that describes the data items to be written.
- ERR=t is optional and specifies the number (t) of an executable statement to which control is to be transferred if a transmission error occurs during the data transfer. Statement t must be in the same program unit as the WRITE statement.
- If ERR is not specified, the preceding comma is omitted.
- The END parameter may not be specified in the WRITE statement.
- list is an I/O list required on unformatted WRITE and optional on formatted WRITE.

Although the ERR parameter need not be specified, if a transmission error occurs, and the ERR parameter is not present, execution of the program will terminate, unless the ERRXIT has been called prior to executing the WRITE statement. (See the discussion of the service subprogram ERRXIT in Appendix B.) ERR must follow the data set reference number and FORMAT statement number, if present.

If the I/O list is specified, it will be treated as one record.

Example:

```
WRITE (2,75) A, B, C
```

This formatted WRITE statement causes data to be written from the variables A, B, and C onto the data set whose reference number is 2 according to the format specified by the FORMAT statement whose number is 75.

Example:

```
WRITE (4) ZEE
```

This unformatted WRITE statement causes data in the variable ZEE to be written onto the data set whose reference number is 4. Since the record is unformatted, no FORMAT statement number is given and none should be specified when the record is read back into storage.

Example:

```
WRITE (5,10,ERR=999) A1,A2
```

This formatted WRITE statement causes data in the variables A1 and A2 to be written onto the data set whose reference number is 5 according to the format specified by the FORMAT statement whose number is 10. If a transmission error occurs during the data transfer, control is transferred to the executable statement numbered 999 (specified by ERR=999 in WRITE statement).

```
WRITE (2,*) I,N(I)
```

This list-directed WRITE statement causes the data in the variable I and the array element N(I) to be written on the device whose data set reference number is 2.

Lists For Transmission Of Data

The list in an input/output statement specifies what elements are to be transmitted. For example, assume that a sequential data set residing on diskette contains the following data:

```
Column 1     7     13    19    25
       |     |     |     |     |
       v     v     v     v     v
       b̸b̸b̸b̸25b̸b̸b̸102b̸b̸-101b̸b̸b̸b̸10b̸b̸b̸b̸b̸5
```

Further assume that the following statement appears in the source program (and 1 specifies the data set reference number):

```
      READ(1,100) I,J,K,L,M
100   FORMAT(5I6)
```

The data set will be read and the program will operate upon the data as though the following statements had been written:

```
I=25
J=102
K=-101
L=10
M=5
```

If control passes back to the READ statement, I, J, K, L, and M will receive new values depending upon what is contained in the next record to be read.

Implied DO Specification in Input/Output Lists

DO-type notation may be used in lists for the transmission of data. For example, suppose it is desired to transmit the five quantities A(1), A(2), A(3), A(4), and A(5). This may be accomplished by writing:

```
10   FORMAT(5F8.0)
12   READ(1,10)(A(I),I=1,5)
```

The above statements cause a record to be read and cause the value contained in the first eight positions of the record to be converted to a real number and stored into A(1), the next eight positions into A(2), etc.
This is equivalent to writing:

```
12   READ(1,10)A(1),A(2),A(3),A(4),A(5)
```

In other words, I would be given the value 1 and the first quantity would become the value of A(1). I would then be increased by 1, and the second quantity would become the value of A(2). This would continue until the fifth quantity to be read becomes the value of A(5).

As with DO statements, a third indexing parameter may be used to specify the amount by which the index is to be incremented at each iteration. Thus,

```
READ(1,50) (A(I),I=1,10,2)
```

causes transmission of values for A(1), A(3), A(5), A(7), and A(9).

General Form of Implied DO Notation

$(y_1, y_2, \ldots, y_n, i=m_1, m_2, m_3)$

where:

- Each y is a list element.
- m_1, m_2 and m_3 are each either an unsigned integer constant or an integer variable. If m_3 is not stated, it is taken to be 1.

As with DOs, i is the index, m_1 is the initial value, m_2 is the test value, and m_3 is the increment. In addition, this notation may be nested.

Example:

```
((C(I,J),D(I,J),I=1,5),J=1,4)
```

would transmit data in the form:

```
C(1,1),D(1,1),C(2,1),D(2,1),...,C(5,1),
D(5,1),C(1,2),D(1,2),...,C(5,4),D(5,4)
```

Additional Details of Input/Output Lists

Any number of quantities may appear in a single list. Integer, real, and logical quantities may be transmitted by the same statement. However, each quantity must have the correct format as specified in a corresponding FORMAT statement if formatted I/O is used.

For formatted READ, only the quantities specified in the list are transmitted; the remaining quantities are ignored. Thus, if a record contains three quantities and a list contains two, the third quantity is not used by the program.

For unformatted READ, a list must not contain more quantities than the input record.

When an array name appears in a list in non-subscripted form, all of the quantities of the array receive data or are transmitted. For example, if A is an array with 25 elements, the statement

```
READ(1,100)A
```

causes all of the quantities A(1),...A(25) to receive data.

A more complex list is:

```
A,B(3),(C(I),D(I,K),I=1,10),
((E(I,J),I=1,10,2),F(J,3),J=1,K)
```

This list would receive or transmit data in the order:

```
A,B(3),C(1),D(1,K),C(2),D(2,K),...,
C(10),D(10,K),E(1,1),E(3,1),
...,E(9,1),F(1,3),E(1,2),E(3,2),...,
E(9,2),F(2,3),...,E(9,K),F(K,3)
```

Note that each item in the list is separated by a comma, that the range of the implied DO statement is clearly defined by means of parentheses, and that constants do not appear in the list except as indexing parameters or subscripts. The variable indexing parameter (K) is assumed to have been previously defined by the program, although in an input list it could have been defined by an item in the list itself, providing that it appeared before its use as an index.

Subscripts appearing in I/O lists may not contain exponentiation (**) or function references.

END FILE Statement

General Form of the END FILE Statement

```
END FILE i
```

where:

- i is an unsigned integer constant or integer variable of length 4 which is a data set reference number. See Appendix C for a description of the NOCMPAT compile option.

The END FILE statement causes an end-of-file record to be written.

Examples:

```
END FILE 10
END FILE K
```

REWIND Statement

General Form of the REWIND Statement

```
REWIND i
```

where:

- i is an unsigned integer constant or integer variable of length 4 which is the data set reference number. See Appendix C for a description of the NOCMPAT compile option.

The REWIND statement causes unit i to be positioned at the first record of the data set.

Examples:

```
REWIND 10
REWIND K
```

BACKSPACE Statement

General Form of the BACKSPACE Statement

```
BACKSPACE i
```

where:

- i is an unsigned integer constant or integer variable of length 4 which is the data set reference number. See Appendix C for a description of the NOCMPAT compile option.

The BACKSPACE statement causes the unit i to backspace one record.

Examples:

```
BACKSPACE 10
BACKSPACE K
```

FORMAT Statement

General Form of a FORMAT Statement

xxxxx FORMAT (c_1 s_1 $c_2 \ldots c_n$)

where:

- xxxxx is a statement number (1 through 5 digits).
- c is a format code (described below).
- s is a separator, which may be either a comma or any number of slashes. Slashes are used to indicate the beginning of a new record. Any number of slashes may precede the first or follow the last format code.

The format codes are:

aIw	Describes integer data fields
aDw.d	Describes double-precision data fields
aEw.d	Describes real data fields
nPaDw.d	Describes double-precision data fields and specifies a scale factor
nPaEw.d	Describes real data fields and specifies a scale factor
aFw.d	Describes real data fields
nPaFw.d	Describes real data fields and specifies a scale factor
aLw	Describes logical data fields
aEw.d	Describes hexadecimal data fields
aAw	Describes character data fields
wH	Describes literal data
literal	Describes literal data
wX	Indicates that a field is to be skipped on input or filled with blanks on output
Tr	Indicates the position in a FORTRAN IV record where transfer of data is to begin
a(. . .)	Indicates a group format specification

where:

- a is optional and is a repeat count, an unsigned integer constant that specifies the number of times the code is to be repeated. If a is omitted, the code is used once.
- w is an unsigned, nonzero integer constant that specifies the number of characters in the field.
- d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point, that is, the fractional portion. The decimal point between w and d portions of the specification is required punctuation.
- n is a negative or unsigned integer constant which is the scale factor; if the constant is unsigned, it is assumed to be positive.
- r is an unsigned integer constant designating a character position in a record.
- (...) is a group format specification. Within the parentheses are format codes, which are separated by commas or slashes.

In order for data to be transmitted from an external storage medium to the computer or from the computer to an external medium, it is necessary that the computer know the form in which the data exists. The FORMAT statement describes the form of the data on the external medium. The FORMAT statement is used with the number of items in the I/O list in the READ and WRITE statements to specify the structure of FORTRAN IV records and the form of the data fields within the records. In the FORMAT statement, the data fields are described with format codes. Delimiters between these format codes specify the structure of the FORTRAN IV records. The I/O list gives the names of the data items in the fields which make up the records. The length of the list, in conjunction with the FORMAT statement, specifies the number of records.

The following are general rules for using the FORMAT statement:

1. FORMAT statements are not executed; their function is to supply information to the object program. They may be placed anywhere in a program unit (other than a BLOCK DATA subprogram) subject to the rules for the placement of the FUNCTION, SUBROUTINE, IMPLICIT, and END statements.
2. Field width may be specified greater than required in order to provide spacing. Thus, if a number is to be converted by I-type conversion and the number does not exceed five characters including sign, a specification of I10 will supply five leading blanks.
3. A specification preceded by an unsigned integer constant may be repeated as many times as desired (within the limits of the device). Thus, FORMAT (2F10.4) is equivalent to FORMAT (F10.4,F10.4).
4. Succeeding specifications are written in a single FORMAT statement separated by a comma. Thus, FORMAT (I2,E10.2) might be used to convert two separate quantities, the first being integer and the second real.
5. To deal with more than one record in a single FORMAT statement, a slash (/) is used to indicate the end of a record. Thus,

   ```
   2 FORMAT (3F9.2,2I12/8E10.5)
   ```

   would specify two records, the first of which would be written according to the format 3F9.22I12 and the second 8E10.5.

   Blank lines may be introduced into the output by the use of consecutive slashes in a FORMAT statement. At the beginning of a FORMAT statement, n consecutive slashes produce n blank lines. In the body of the statement, n consecutive slashes produce n−1 blank lines; at the end of the statement, n slashes produce n blank lines.

   Consecutive slashes in a FORMAT statement on input cause records to be skipped. The number of records skipped is determined by the position of the slashes in the FORMAT statement. Thus, at the beginning and at the end of the FORMAT statement, n consecutive slashes cause n records to be skipped. In the body of the FORMAT statement, n consecutive slashes cause n−1 records to be skipped.

6. When formatted records are prepared for printing, the first character of the record can be treated as a carriage control character. By specifying this option, certain printer controls are available. The carriage control character is generally specified with a literal format code: either c or 1Hc, where c is one of the following:

c	*Meaning*
blank	Advance one line before printing
0	Advance two lines before printing
1	Advance to first line of next page
+	No advance

If the carriage control option has been specified, the first character of the record is not interpreted as part of the data field but is always used for carriage control, whether or not it is specified directly in the FORMAT statement. Thus, formatting records without considering the carriage control may cause problems. For example, if the first field of a record is numeric and produces a 1 as the first character of the record, it will cause an advance to the first line of the next page on the printer.

Format codes that may affect carriage control are listed below:

- literals: (no associated I/O list item)

Example:

```
    WRITE (2,100) I
100 FORMAT ('1PAGE',I5)
```

These statements position a 1 in the first character of the record, causing a page eject.

- T- and X-format codes: (no associated I/O list item). These formats can produce leading blanks in the first character of a record. Since the first character is not printed, it is necessary, when using the T-format code, to tab to n+1 to start printing in the nth print position. For example, to start printing in print position 3, the T code is T4.

Example:

```
    WRITE
10  FORMAT (T61, 'TABULATING')
```

These statements would cause the word TABULATING to be printed in positions 60 through 69. The 61 leading blanks include a blank in the first position of the record, which causes the carriage to advance one line before printing.

Example:

```
    WRITE (2,15) I,J
15  FORMAT (20X,2I10)
```

These statements would cause the carriage to advance one line and then print the two integer fields defined by I and J in positions 20 through 40.

- A-conversion: (with associated I/O list item)

Example:

```
    DATA ICARR/'+'/
    WRITE (2,20) ICARR,AX
20  FORMAT (A1,F10.2)
```

These statements cause no advance of the printer before printing the value of the real variable AX in print positions 1–10.

- I-conversion: (for 0, 1, and leading blanks)

Example:

```
    WRITE (2,30) I,J
30  FORMAT (I4,I6)
```

where I=14739

These statements cause the printer to advance to a new page before printing 473 in print positions 1–3 followed by the value of the integer variable J.

- E- or F-conversion: (with a field long enough to produce leading blanks or with a shorter field to produce a 0 or 1 in the first position)

Example:

```
    WRITE (2,20) A1,A2
20  FORMAT (F8.2,F10.4)
```

where A1=7678.2

These statements cause the printer to advance to a new line because a blank is in the first position of the A1 field and then prints 7678.20 in print positions 1–7 followed by the value of A2.

- Z-conversion: (with a field that produces leading blanks, or a leading 0 or 1)

Example:

```
    DATA HEX/Z04BC61D2/
    WRITE (2,70) HEX
70  FORMAT (Z8)
```

These statements cause the printer to advance two lines before printing 4BC61D2 in print positions 1–7.

7. The specifications in a FORMAT statement must have correspondence in type with the items in the input/output statements: integer quantities require I-conversion, REAL*4 quantities require E- or F-conversion, REAL*8 quantities require D-conversion, and logical quantities require L-format code. A- or Z-conversions can correspond to any type in the I/O list.
Thus, the following statements are compatible:

```
    WRITE (3,2) A,B,I
2   FORMAT (2F6.4,I10)
```

The following statements are incompatible:

`    WRITE(3,2)A,B,C,`	A, B, and C are real variables
`2   FORMAT(2F6.4,I10)`	I specifies integer conversion

8. When defining a FORTRAN IV record by a FORMAT statement, it is important to consider the maximum size record allowed on the input/output medium. For example, if a 78-character record is to be written, the FORMAT statement must not define a record longer than 78 characters. If the record is to be printed on the programmer console, its length must not be longer than the keyboard's line length. For input, the FORMAT statement must not define a FORTRAN IV record longer than the actual input record.
9. Successive items in the input/output list are transmitted by successive corresponding specifications in the FORMAT statement until all items in the list are transmitted. If there are more items in the list than there are specifications, the record is ended and control transfers to the last preceding left parenthesis of the FORMAT statement for the next record. This will be either the left parenthesis at the beginning of the FORMAT statement or, if grouping was used, the left parenthesis of the last group in the FORMAT statement.

For example, suppose the following statements are written into a program:

```
    WRITE (3,10) A,B,C,D,E,F,G
10  FORMAT (F10.3,E12.6,F12.2)
```

Then the following table shows the variable transmitted in the column on the left and the specification by which it is converted in the column on the right.

Variable Transmitted	*Specification*	
A	F10.3	
B	E12.6	first record
C	F12.2	
D	F10.3	
E	E12.6	second record
F	F12.2	
G	F10.3	third record

If the FORMAT statement is coded

```
10 FORMAT (F10.3,E12.6,2(F12.2))
```

the results would be as follows:

Variable Transmitted	*Specification*	
A	F10.3	
B	E12.6	first record
C	F12.2	
D	F12.2	
E	F12.2	second record
F	F12.2	
G	F12.2	third record

10. A limited grouping by parentheses is permitted in order to enable repetition of data fields according to certain format specifications within a longer FORMAT statement specification. Thus, FORMAT (2(E10.5,E12.6),I4) is equivalent to FORMAT (E10.5,E12.6,E10.5,E12.6,I4). An additional level of parentheses is not permitted. Thus, FORMAT (2(3(I6,I8))) is invalid. However, FORMAT (2(I2),2(I4)) is valid because additional parentheses are invalid only within group parentheses.
11. The maximum value of a repeat factor, whether of the form aEw.d or a(Ew.d), is 255 characters.
12. Numeric input data to be read by means of a READ statement when the object program is executed must be in the same format as given in the previous examples. Thus, a record to be read according to FORMAT (I2,E12.4,F10.4) might be created:

    ```
    27ƀ-0.9321Eƀ02ƀƀƀ-0.0076
    ```

    Within each field, all information is taken to be right justified; embedded blanks and trailing blanks in numeric fields are read as zeros and will affect the item's value. Plus signs may be omitted or indicated by a +. Minus signs must be present if the number is negative or has a negative exponent. Certain variations in input data format are permitted.

    - Numbers of D- and E-conversion need not have 4 columns devoted to the exponent field. The start of the exponent field must be marked by a D or E, or if that is omitted, by a + or – (not a blank). Thus, E2, E+2, +2, +02, E02, and E+02 are all permissible exponent fields.
    - Numbers for D-, E-, and F-conversion need not have decimal points. If they are not written, the format specification will supply them. For example, the number –69321 + 2 with the code E12.4 will be treated as though the decimal point had been supplied between the 6 and the 9. If the decimal point is supplied, its position overrides the position indicated in the FORMAT statement.

Example:

```
    WRITE (1,5) I,A,J
5   FORMAT (I5,F8.4/20X,I5)
```

This format specifies two records. The first record would be written according to the format I5,F8.4 and describes two fields. The first field allows five columns for an integer and the second field allows eight columns (four following the decimal point) for a real number.
The second record is written according to the format 20X,I5. This causes 20 blanks to precede a five-digit integer.

Example:

```
    WRITE (1,10) NUM
10  FORMAT (' THE ANSWER IS ',I10)
```

This causes the output to be written THE ANSWER IS and allows a ten-digit integer to follow.

Example:

```
    READ (1,33) A,I,B,C,J,D,E
33  FORMAT (F6.2,2(I3,2F2.1))
```

This format defines a record in which the first six columns contain a real number (three before the decimal point, one for the decimal point, two following the decimal point), followed by a three-column integer and two real numbers of length two (one column for the decimal point and one column following the decimal point), a three-column integer and two real numbers of length two.

Example:

```
    WRITE (2,100,ERR=999) A,B,C
100 FORMAT (1H1,2F10.3/'0',E12.4)
```

This format defines two records that will be printed (assuming the 2 in the WRITE statement corresponds to the printer) using the carriage control character option. The first record consists of two fields preceded by a carriage control field containing the character 1. This carriage control character causes the printer to advance to the first line of the next page and then print the two fields described by A and B, according to the format 2F10.3. This format causes ten columns to be printed for each field (six before the decimal point, one for the decimal point, three following the decimal point).

The second record is also preceded by a carriage control character. Designating 0 as the first character of the record causes the printer to advance two lines before printing the real number C in the format E12.4. This format allows twelve columns to be printed (three before the decimal point, one for the decimal point, four following the decimal point, and four for the exponent).

Conversion of Numeric Data

Four types of conversion for numeric data are:

Internal	*Conversion Code*	*External*
Real	F	Real (without exponent)
Real	E	Real (with exponent; single precision)
Real	D	Real (with exponent; double precision)
Integer	I	Integer

Numbers printed by F-conversion are printed as output in a decimal notation without an exponent. Typical output might be:

12.3	-0.726	102.
-17.2	1.318	-968.
289.1	0.009	721.

Numbers printed by D- and E-conversion are printed as a decimal number with a power of 10. If the scale factor equals zero, these numbers are normalized; that is, their first significant digit is to the right of the decimal point. For example:

232.3	is printed as	0.2323E+03
.003	is printed as	0.30E-02
17.4	is printed as	0.174E+02

For Fw.d format specifications, w should exceed d by at least 1 for positive numbers and 2 for negative numbers; for Ew.d, w should exceed d by at least 5 for positive numbers and 6 for negative numbers.

Numbers printed by I-conversion are printed as integers. Typical output might be:

12
-17
2342

Programming Note: If an integer input field (I format) contains a number whose absolute value exceeds 2,147,483,647, or if a real input field (E-, F-, or D-format) contains a number whose absolute value is less than $.539 \times 10^{-79}$ or greater than $.723 \times 10^{75}$, the results are unpredictable.

No error message is issued for any of the above-mentioned error cases, but the user can determine if an error has occurred by calling FCTST. If incorrect data was entered, FCTST will return error code 128. The user may then take additional action within his program.

If the mantissa of a single-precision input number exceeds 9 digits or the mantissa of a double-precision input number exceeds 18 digits, the least significant digits will be truncated and the exponent will be adjusted to reflect the number of digits truncated. This is not considered an error condition.

I-Conversion (aIw)

I-conversion must be used to read integer data or to print a number which exists in the computer as an integer quantity.

Input: w characters are read from an input device. Leading, embedded, and trailing blanks are treated as zeros. If the number is too large to be contained in an INTEGER*2 or INTEGER*4 variable, only the leftmost digits are used, and computations involving this variable will be meaningless.

The following examples show the internal values of the given quantities if read under the I3 format code:

External Form	*Internal Value*
3ƀƀ	300
ƀƀƀ	0
ƀ3ƀ	30
ƀ−2	−2

Output: w print positions are reserved for the number. It is printed in a w-space field right-justified (that is, the units position is at the extreme right). If the number converted is greater than w positions, asterisks are printed instead of the number. If the number has fewer than w digits, the leftmost spaces are filled in with blanks. If the quantity is negative, the space preceding the leftmost digit will contain a minus sign; w must be large enough to allow a position for this minus sign.

The following examples show how each of the quantities on the left is printed according to the specification I3 (ƀ is used to indicate blanks):

Internal Value	*Printed Form*
721	721
−721	***
−12	−12
9	ƀƀ9
8114	***
0	ƀƀ0
−5	ƀ−5

D- and E-Conversion (aDw.d), (aEw.d)

D- and E-format codes are used to transmit real or double-precision data.

Input: The number may have a decimal point, a D, an E, or a signed integer constant exponent. Exponents must be preceded by a constant, that is, an optional sign followed by at least one decimal digit (with or without a decimal point). If the decimal point is present, its position overrides the position indicated by the d portion of the format specification. In addition, the number of positions specified by w must include a place for it. Since leading, trailing, and embedded blanks are treated as zeros, any embedded and trailing blank will affect the value of the item.

The D, E, and signed integer constant exponent specifications for input data are interchangeable. For example, given a REAL*4 item in an input list and E format specification, the exponent specification in the data item may be a D, an E, or a signed integer constant, or have no exponent. The data item will be treated as a REAL*4 constant in any case. Similarly, if the list item is REAL*8 and the FORMAT specification is D, the data item will be treated as a double-precision constant regardless of its exponent specification, if any. Note that the type and length of the list item must agree with that of the specification.

Output: Unless a scale factor is present (the scale factor changes the location of the decimal point in real numbers and its use is explained later in this section), output consists of an optional sign (required if the value is negative), a decimal point, the number of significant digits specified by d, and a D or E exponent requiring four positions: the D or E, a + or − sign, and a two-digit exponent. The w specification must provide spaces for all of these positions. Thus, its value should always be at least d + 5, or d + 6 if the number can be negative. If additional space is available, a leading zero will be written before the decimal point. If the value of w is not sufficient to print a decimal point and a four-position exponent (and a minus sign if the value is negative), asterisks will be printed instead of the number. Fractional digits in excess of the number specified by d are dropped after rounding.

Example:

```
E10.5
```

This format, on input, causes the following:

+56789.0E2	is converted to internal equivalent of	5678900.
-5678934E5	is converted to internal equivalent of	-5678934.
5678.900E0	is converted to internal equivalent of	5678.900
567891E-01	is converted to internal equivalent of	.567891
+567.893+3	is converted to internal equivalent of	567893.
+56.789E-3	is converted to internal equivalent of	.056789
+56789.100	is converted to internal equivalent of	56789.1

F-Conversion (aFw.d)

F-conversion is used to transmit real data fields.

Input: w is the total field width including the exponent, if any, and d is the number of places to the right of the decimal point (the fractional portion). If a decimal point is present in the data, its position overrides the d specification in the format code. Either an E or a signed integer exponent is acceptable as input with an F-format code. Blanks are treated as zeros; thus, embedded and trailing blanks will affect the value of the number.

For example, the following items will be interpreted as having the value 1000:

Field Description	*Input Record*
F8.0	ƀƀƀƀ1ƀƀƀ
F8.6	ƀƀƀ1ƀƀƀ.

Output: w must provide sufficient space for the integer part if it is other than zero, a fractional part containing d digits, a decimal point, and, if the output value is negative, a sign. Thus, the value of w should be at least 1 greater than the value of d and at least 2 greater if the number can be negative. If insufficient positions are provided for the sign (if minus), integer portion, decimal point, and d-digit fraction, asterisks are written instead of the number. If excess positions are provided, the number is preceded by blanks. Fractional digits in excess of the number specified by d are dropped after rounding.

The following example shows how each of the quantities on the left is printed according to the specification F5.2:

Internal Value	*Printed Form*
12.125	12.13
−41.5	*****
−0.25	−0.25
7.375	ƀ7.38
−1.	−1.00
9.03125	ƀ9.03
187.625	*****
0.00390625	ƀ0.00
0.0078125	ƀ0.01

Scale Factor (nPaDw.d, nPaEw.d, or nPaFw.d)

The P scale factor may be specified as the first part of a D, E, or F field descriptor to change the location of the decimal point in real numbers.

Unless there is an exponent in the external input or output data field, the effect of the scale factor for F-conversion is external number = internal number x 10^n, where n is the scale factor—the number preceding P.

Input: The scale factor in the format specification is ignored for any data item with an exponent in the external field. Otherwise, a positive scale factor decreases the magnitude of the data item and a negative scale factor increases its magnitude. For example, if the input data is in the form xx.xxxx and is to be used internally in the form .xxxxxx, then the format code used to effect this change is 2PF7.4. Or, if the same input is to be used in the form xxxx.xx, then the format code used to effect this change is −2PF7.4.

Output: The scale factor can be specified for F-, D-, or E-conversion. For F-conversion, the effect of the scale factor is the opposite of that for input; a positive scale factor increases the magnitude of the number and a negative scale factor decreases the magnitude. For example, if the number has the internal form xx.xxxx and it is to be written out in the form xxxx.xx, the format code used to effect this change is 2PF7.2.

For D- or E-conversion, the exponent is adjusted so that the magnitude of the number does not change. For example, if the internal number 238.50 is printed according to the format E10.3, it appears as 0.238E+03. If it is printed according to the format 1PE10.3, it appears as 2.385E+02.

Once a scale factor has been established, it applies to all subsequently interpreted D-, E-, and F-format codes in the FORMAT statement until another scale factor is established. A factor of 0 may be used to discontinue the effect of a previous scale factor. If no scale factor is given, 0 is used for all F-, D-, or E-conversion.

Example:

```
30 FORMAT (E8.3,2PE10.4,E7.2,I5)
```

No scale factor applies to E8.3. The scale factor 2 applies to both the E10.4 and the E7.2 specifications. If five data items were read using this code, scale factor 2 would also apply to the E8.3 code, which would be used to interpret the fifth item. To discontinue the effect of the code after the E10.4 specification, the statement should be coded:

```
30 FORMAT (E8.3,2PE10.4,0PE7.2I5)
```

To discontinue it after the E7.2 code if more than four data items are to be read using the FORMAT statement, the statement should read:

```
30 FORMAT (0PE8.3,2PE10.4,E7.2,I5)
```

Note that the 0PE8.3 specification is not necessary to discontinue the effect of a scale factor in a previous FORMAT statement or in the previous use of this FORMAT statement.

L-Format Code (2Lw)

The L-format code is used in transmitting logical variables.

Input: The input field in the data consists of optional blanks, followed by a T or F, followed by optional characters for true or false, respectively. The T or F causes a value of true or false to be assigned to the logical variable in the input list.

For example, assume the following statements:

```
25   FORMAT (4L8)
     READ (3,25) Q1,Q2,Q3,Q4
```

and the input data:

```
       ▽       ▽       ▽       ▽
       8      16      24      32
b̸b̸b̸b̸b̸b̸b̸Tb̸b̸b̸FALSEb̸b̸b̸b̸b̸b̸b̸Fb̸b̸b̸TRUE
       |       |       |       |
```

Then logical variables Q1 and Q4 would be given the values true and Q2 and Q3 the values false.

Output: A T or F is inserted in the output record depending upon whether the value of the logical variable in the I/O list is true or false, respectively. The single character is right-justified in the output data field and preceded by w-1 blanks.

Z-Conversion (aZw)

The Z-format code is used in transmitting hexadecimal data.

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. Therefore, if an input field contains an odd number of digits, the number is padded on the left with a hexadecimal zero 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 hexadecimal digits in the variable is less than w, the leftmost print positions are filled with blanks. If the number of characters in the storage location is greater than w, the leftmost digits are truncated and the rest of the number is printed.

Example:

```
25 FORMAT (4Z5)
```

specifies four hexadecimal fields containing five columns each.

Examples of Numeric Format Codes

The following examples illustrate the use of the format codes I, F, E, D, and Z.

Example:

```
75   FORMAT (I3,F5.2,2E10.3)
     READ (5,75) N,A,B,C
```

- A record containing four input fields is described in the FORMAT statement and four variables are in the I/O list. Therefore, each time the READ statement is executed, one input record is read from the data set associated with data set reference number 5.
- When an input record is read, the number from the first field of the record (three columns) is stored in integer format in location N. The number from the second field of the input record (five columns) is stored in real format in location A. The next two numbers from the third and fourth fields are stored in real format in locations B and C.
- If one more variable (M, for example) were added to the I/O list, another record would be read and the number from the first three columns of that record would be stored in integer format in location M. The rest of the record would be ignored.
- If one variable (C, for example) were deleted from the list, one format specification E10.3 would be ignored.

Example:

```
75   FORMAT (Z4,D10.3,2F8.3)
     READ (5,75) A,B,C,D
```

where A, C, and D are REAL*4 and B is REAL*8 and, on successive executions of the READ statement, the following input records are read:

```
Column:   1   5         15      23      31
          |   |         |       |       |
          v   v         v       v       v
Input     b̸3F1156432D+02b̸b̸123.4245781315
records   2AF3155381+02b̸-b̸8756191 46.7345
          3ACb̸346.18D-03b̸b̸+.145614.67345
Format:   Z4, D10.3     F8.3    F8.3
```

then the variables A, B, C, and D receive values as if the following data had been read:

A	*B*	*C*	*D*
03F1	156.432D02	123.42	45781.315
2AF3	155.381+20	−875.619	146.7345
3ACO	346.18D-03	.1456	14.67345

- Leading, trailing, and embedded blanks in an input field are treated as zeros. Therefore, since the value for B on the second input card was not right justified in the field, the exponent is 20 rather than 2.
- If an explicit decimal point appears in the input field, it overrides the d field, of the Ew.d, Dw.d, or Fw.d specification.

Example:

```
76   FORMAT (' ',F6.2,E12.3,I5)
     WRITE (6,76),A,B,N
```

where the variables A, B, and N are the internal equivalent of the following values on successive executions of the WRITE statement:

A	*B*	*N*
34.40	123.380E+02	31
31.	1156.1E+02	130
−354.32	834.621E-04	428
1.132	83.121E+06	

then, the following records are created by successive executions of the WRITE statement:

```
Column:       7             19     24
              ↓              ↓      ↓
          34.40     0.123E+05      31
          31.10     0.116E+06     130
         ******     0.835E+00     428
           1.13     0.831E+08       0
```

- The integer portion of the third value of A exceeds the format specification, so asterisks are printed instead of a value. The fractional portion of the fourth value of A exceeds the format specification, so the fractional portion is rounded.
- Note that, for the variable B, the decimal point is printed to the left of the first significant digit and that only three significant digits are printed because of the format specification E12.3. Excess digits are rounded off from the right.

Handling of Alphameric Data

There are three specifications available for input/output of alphameric information: A-conversion, H-conversion, and literals enclosed in apostrophes.

A-Conversion (aAw)

The specification aAw causes w characters to be read into, or written from, a variable or array element. The type of the variable or array is immaterial, since no conversion takes place. Thus, the A-format code can be used for numeric fields but not for numeric fields requiring arithmetic. The maximum width of w is 255.

Input: The number of characters stored in internal storage depends on the length of the variable in the I/O list. If w is greater than the variable length (v, for example) then the leftmost w-v characters in the field of the input card are skipped and the remaining v characters are read and stored in the variable; truncation occurs on the left. If w is less than v, then w characters from the field in the input card are read and the remaining rightmost characters in the variable are filled with blanks.

Output: If w is greater than the length (v) of the variable in the I/O list, then the output field will contain v characters right-justified in the field and preceded by leading blanks. If w is less than v, the leftmost w characters from the variable will be printed and the rest of the data will be truncated; truncation occurs on the right.

H-Conversion (wH) and Literals Enclosed in Apostrophes

The specification wH is followed in the FORMAT statement by a string of w alphameric characters. For example:

```
24HƀTHISƀISƀALPHAMERICƀDATA
```

This specification may also be coded using apostrophes to enclose the string of characters:

```
'ƀTHISƀISƀALPHAMERICƀDATA'
```

The apostrophe specification method may be more convenient for specifying long character strings.

The count begins with the character immediately following the H. Note that blanks are considered alphameric characters and must be included as part of the count w.

The effect of wH or literal specification depends on whether it is used with input or output.

Input: w characters, or as many characters as are enclosed in apostrophes, are extracted from the input record and replace the characters written in the FORMAT statement.

Output: The string of characters following the specification or the literal string is written as part of the output record unless characters have replaced them as a result of an input operation, in which case the replacement characters are written.

For example, suppose that the following statements are executed:

```
  WRITE (3,2)
2 FORMAT (20HTIME/QUANTITYƀREPORT)
```

These would cause the following output to be written:

```
TIME/QUANTITY REPORT
```

Now assume that a record containing the characters ƀNOƀƀ238 is read using these statements:

```
  READ (1,1)I
1 FORMAT ('YES',I5)
```

The statement

```
WRITE (3,1)I
```

would create the following output:

```
ƀNOƀƀ238
```

Skipping Fields in a Record (X-Format Code)

Blank characters may be provided in an output record or characters of an input record may be skipped by using the specification wX, where w is the number of blanks provided or characters skipped.

For example, if a record has six 10-column fields for integers and it is not desired to read the second quantity, then the statement

```
10 FORMAT (I10,10X,4I10)
```

may be used along with the appropriate READ statement. The T-format code can also be used for this purpose, as described below.

Tabulating the Record (T-Format Code)

The FORMAT statement processes the data in a record from left to right, according to the specifications given within the FORMAT statement. Often, it may be useful to start writing in other than record position 1 or to write data at specific record locations. The T-format code can be used for this purpose.

The T-format code is specified as

```
Tr
```

where r is an unsigned integer constant specifying the position in the record where data transfer is to begin.

A blank is inserted into any character position that has not been previously filled.

Example:

```
1 FORMAT (T61,'CALCULATION')
```

This statement would cause the word CALCULATION to be inserted in positions 61 through 71 of the external record and positions 1 through 60 to be filled with blanks.

Example:

```
2 FORMAT (T21,E20.7)
```

This statement would cause the data item whose specification is E20.7 to be inserted starting at position 21.

More than one T specification may be used in a FORMAT statement. The print positions specified need not be sequential.

Example:

```
3 FORMAT (T2,'MINUS',T40,'PLUS')
```

This statement would cause MINUS to be written starting at position 2 and PLUS to be written starting at position 40. Position 1 and positions 7 through 39 would be blank.

The positions of all codes following a T specification are governed by that specification until another T specification is encountered.

Example:

```
5 FORMAT (T1,'ANS=',T20,E9.3,T35,F10.3,I10,'NOTE 1')
```

This statement would result in the following line being written:

```
ANS =           0.354E 02       111082.986             536453 NOTE 1
^               ^               ^          ^                  ^
:               :               :          :                  :
Position        Position        Position   Position           Position
1               20              35         45                 55
```

When formatting a line using the T specification, care must be taken not to overlap print positions.

The T specification can also be used for input.

Example:

```
    READ (1,6) INPUT
6   FORMAT (T15,I5)
```

These statements would cause five columns of the input record, beginning at column 15, to be transmitted to the variable INPUT.

If you specify the carriage control option, the first character of the record is not treated as data but is interpreted as the carriage control character. (It is necessary to tab to n+1 to start printing in the nth print position.)

Example:

```
    WRITE (2,10)
10  FORMAT (T61,'TABULATING')
```

These statements would cause the word TABULATING to be inserted in print positions 60 through 69.

List-Directed Input Data

A record containing list-directed input data consists of an alternation of constants and separators. The record may be read only from a sequential data set; for example, a keyboard device.

An input constant may be of any valid FORTRAN IV numeric data type. Blanks may not be embedded in any list-directed constant since they would be interpreted as separators. Numeric constants may optionally be signed, but there must be no embedded blanks between the sign and the constant.

Each constant must agree in type with the corresponding list element. The decimal point may be omitted from a real constant. If omitted, it is assumed to follow the rightmost digit of the constant.

With the exceptions noted below, a separator is either a comma or a blank. In addition, for console input, an end indicator is a separator. Blanks may optionally occur between the comma and the carriage return or end-of-card.

A separator may be surrounded by any number of blanks, horizontal tabs, or carriage returns. Any such combination (with no intervening constants) constitutes a single separator. When execution of a list-directed READ statement begins, a preceding separator is assumed and initial blanks, horizontal tabs, or carriage returns, if present, are considered part of that separator.

A null item is represented by two consecutive commas with no intervening constant. Any number of blanks, horizontal tabs, or carriage returns, may be embedded between the commas. If a null item is specified, the corresponding list item is skipped; its current value remains unaltered.

A repeat factor may be specified for a constant or null item. For a constant, the form is

```
i*constant
```

and for a null item, the form is

```
i*
```

In each instance, i is a nonzero, unsigned integer constant, indicating that the following constant or null item is to occur i times. Neither of these forms may contain embedded blanks. The separators surrounding a repeated null item need not be commas.

A slash (/) serves as a special-purpose separator, indicating that no more data is to be read during the current execution of a READ statement. If the list has not been satisfied, the values of the remaining list elements are unaltered. If the list has been satisfied, the slash is optional.

List-Directed Output Data

List-directed output data may be directed to any unit-record output device such as a printer and may contain any producible form of data which is readable as list-directed input. However, certain forms which are permissible as list-directed input are not produced as list-directed output. These forms are: null items, i* repeat factor, and / special-purpose separator.

In list-directed output, the width of the data field depends upon the type of variable to be written.

Type of Variable	*Width of Data Field*
REAL*8	24 characters
REAL*4	14 characters
INTEGER*4	11 characters
INTEGER*2	6 characters

A blank is inserted as a separator between data fields. The total width of the data fields plus separators must be considered when output is going to unit-record devices.

Direct-Access Input/Output Statements

The direct-access statements permit you to read and write records randomly from any location within a data set. They contrast with the sequential input/output statements, which process records from the beginning of a data set to its end. With the direct-access statements, you can go directly to any point in the data set, 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: DEFINE FILE, READ, WRITE, and FIND. DEFINE FILE is a non-executable statement which describes the characteristics of the data sets to be used during a direct-access operation. The FIND statement is used to point to the next record required. The READ and WRITE statements cause transfer of data into or out of internal storage. These statements allow you to specify the location within a data set from which data is to be read or into which data is to be written.

In addition to these four statements, the FORMAT statement (described previously) specifies the form in which data is to be transmitted. All data set reference numbers referenced by direct-access READ, WRITE, and FIND statements must be defined by a DEFINE FILE statement.

Each record in a direct-access file has a unique record number associated with it. You must specify in the READ, WRITE, and FIND statements not only the data set reference 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 data set 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 data set.

DEFINE FILE Statement

To use the direct-access READ, WRITE, and FIND statements in a program, the data sets to be operated on must be described with DEFINE FILE statements. Each direct-access data set must be described once and only once in the main program.

General Form of the DEFINE FILE Statement

$\texttt{DEFINE FILE } u_1(r_1,s_1,f_1,v_1),$
$u_2(r_2,s_2,f_2,v_2),\ldots,$
$u_n(r_n,s_n,f_n,v_n)$

where:

- Each u is an unsigned integer constant that is the data set reference number.
- Each r is an integer constant that specifies the number of records in the data set associated with u.
- Each s is an integer constant that specifies the maximum size of each record associated with u. The record size is measured in bytes or doublewords (four bytes). The method used to measure the record size depends upon the specification for f.
- Each f specifies that the data set is to be read or written either with or without format control; f may be one of the following letters:
  - L to indicate that the data set 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 to indicate that the data set 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 bytes.
  - U to indicate that the data set is to be read or written without format control and that the maximum record size is measured in number of doublewords.
- Each v is an integer variable (not an array element) called an associated variable. 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 must be set to a value prior to the first read or write operation on the data set.

The associated variable cannot appear in the I/O list of a READ or WRITE statement for a data set with which it is associated.

Example:

```
DEFINE FILE 8(50,100,L,I2),9(100,50,L,J3)
```

This DEFINE FILE statement describes two data sets, identified by reference numbers 8 and 9. The data in the first data set consists of 50 records, each with a maximum length of 100 bytes. 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 data in the second data set consists of 100 records, each with a maximum length of 50 bytes. 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.

If an E is substituted for each L in the preceding DEFINE FILE statement, a FORMAT statement is required and the data is transmitted under format control. If the data is to be transmitted without format control, the DEFINE FILE statement can be written as:

```
DEFINE FILE 8(50,25,U,I2),9(100,13,U,J3)
```

Direct-Access Programming Considerations

When programming for direct-access input/output operations, you must establish a correspondence between FORTRAN IV records and the records described by the DEFINE FILE statement. All conventions discussed in the section "FORMAT Statement" apply.

For example, to process the data set described by the statement

```
DEFINE FILE 8(10,48,L,K8)
```

the FORMAT statement used to control the reading or writing could not specify a record longer than 48 bytes. The statements

```
FORMAT(4F12.1)
FORMAT(I12,9F4.2)
```

define a FORTRAN IV record that corresponds to those records described by the DEFINE FILE statement. The records can also be transmitted under format control by substituting an E for the L and rewriting the DEFINE FILE statement as:

```
DEFINE FILE 8(10,48,E,K8)
```

To process a direct-access data set without format control, the number of storage locations specified for each record must be greater than or equal to the maximum number of storage locations in a record to be written by any WRITE statement referring to the file. For example, if the I/O list of the WRITE statement specifies transmission of the contents of 100 bytes (25 doublewords), the DEFINE FILE statement can be either of the following:

```
DEFINE FILE 8(50,100,L,K8)
DEFINE FILE 8(50,25,U,K8)
```

Programs may share an associated variable as a COMMON or GLOBAL variable. The following example shows how this can be accomplished.

```
  COMMON IUAR
  DEFINE FILE 8(100,10,L,IUAR)
  •
  •
  •
  ITEMP=IUAR
  CALL SUBI(ANS,ARG)
4 IF (IUAR-ITEMP) 20,16,20
  •
  •
  SUBROUTINE SUBI(A,B)
  COMMON IUAR
  •
  •
  •
```

In this example, the program and the subprogram share the associated variable IUAR. An input/output operation that is performed on data set number 8 and is performed in the subroutine causes the value of the associated variable to be changed. The associated variable is then tested in the main program in statement

4. An associated variable should be passed to a subprogram in a CALL statement.

READ Statement

The READ statement causes data to be transferred from a direct-access device into internal storage. The data set being read must be defined with a DEFINE FILE statement.

General Form of the Direct-Access READ Statement

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

where:

- u is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; u must be followed by an apostrophe ('). See Appendix C for a description of the NOCMPAT compile option.
- r is an integer expression that represents the relative position of a record within the data set associated with u. The relative record number of the first record of a direct-access data set is 1.
- f is optional and, if given, specifies the statement number of the FORMAT statement that describes the data being read.
- ERR=s is optional and s is the number of an executable 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.
- list is an I/O list and is optional.

The I/O list must not contain the associated variable defined in the DEFINE FILE statement for data set u.

Example:

```
     DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)

     DIMENSION M(10)
     •
     •
     •
     ID2 = 21
     •
     •
     •
10   FORMAT (5I20)
 9   READ (8'16,10) (M(K),K=1,10)
     •
     •
     •
13   READ (9'ID2+5) A,B,C,D,E,F,G
```

READ statement 9 transmits data from the data set, identified by reference number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 bytes each are read as specified by the I/O list and FORMAT statement 10. Two records are read to satisfy the I/O list, because each record, as defined by the FORMAT statement, contains only five data items (100 bytes). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

READ statement 13 transmits data from the data set, identified by reference number 9, without format control; record 26 is read, and data is transmitted until the I/O list for statement 13 is satisfied. Because the DEFINE FILE statement for data set 9 specified the record length as 28 bytes, the I/O list of statement 13 calls for the same amount of data (the seven variables are type real and each occupies four-bytes). The associated variable ID2 is set to a value 27 at the conclusion of the operation. If the value of ID2 is unchanged, the next execution of statement 13 reads record 32.

The DEFINE FILE statement in the previous example can also be written as:

```
DEFINE FILE 8(500,100,E,ID1),9(100,7,U,ID2)
```

The FORMAT statement may also control the point at which reading starts. For example, if statement 10 in the example is

```
10 FORMAT (//5I20)
```

records 16 and 17 are skipped, record 18 is read, records 19 and 20 are skipped, record 21 is read, and ID1 is set to a value of 22 at the conclusion of the read operation in statement 9.

WRITE Statement

The WRITE statement causes data to be transferred from internal storage to the disk. The data set being written must be defined with a DEFINE FILE statement.

General Form of the Direct-Access WRITE Statement

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

where:

- u is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; u must be followed by an apostrophe ('). See Appendix C for a description of the NOCMPAT compile option.
- r is an integer expression that represents the relative position of a record within the data set associated with u.
- f is optional and, if given, specifies the statement number of the FORMAT statement that describes the data being written.

- ERR=s is optional and s is the number of an executable statement in the same program unit as the WRITE statement to which control is given when a device error condition is encountered during data transfer from storage to device.*
- list is an I/O list. It is optional if f is specified.

*The ERR parameter need not be specified; however, if a transmission error occurs and the ERR parameter is not coded, execution of the program may be terminated. (See the discussion of the service program ERRXIT in Appendix B.) If coded, ERR must follow the data set reference number, the relative record number, and the FORMAT statement number.

The I/O list must not contain the associated variable defined in the DEFINE FILE statement for data set u.

Example:

```
    DEFINE FILE 8(500,100,L,ID1),9(100,28,L,ID2)
    DIMENSION M(10)
    •
    •
    •
    ID2=21
    •
    •
    •
10  FORMAT (5I20)
 8  WRITE (8'16,10) (M(K),K=1,10)
    •
    •
    •
11  WRITE (9'ID2+5) A,B,C,D,E,F,G
```

WRITE statement 8 transmits data into the data set, identified by reference number 8, under control of FORMAT statement 10; transmission begins with record 16. Ten data items of 20 bytes each are written as specified by the I/O list and FORMAT statement 10. Two records are written to satisfy the I/O list because each record contains 5 data items (100 bytes). The associated variable ID1 is set to a value of 18 at the conclusion of the operation.

WRITE statement 11 transmits data into the data set, identified by reference number 9, without format control; transmission begins with record 26. The contents of 14 words are written as specified by the I/O list for statement 11. The associated variable ID2 is set to a value of 27 at the conclusion of the operation. Note the correspondence between the records described (14 words per record) and the number of items called for by the I/O list (7 variables, type real, each occupying four-bytes).

The DEFINE FILE statement in the example can also be written as:

```
DEFINE FILE 8(500,100,E,ID1),9(100,7,U,ID2)
```

As with the READ statement, a FORMAT statement may also be used to control the point at which writing begins.

FIND Statement

The FIND statement causes the associated variable to be updated to the value of the record number in the FIND statement. The record is brought into main storage and made available so that a later READ statement may more quickly fill an I/O list.

General Form of the FIND Statement

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

where:

- u is an unsigned integer constant or an integer variable (not an array element) that is of length 4 and represents a data set reference number; u must be followed by an apostrophe ('). See Appendix C for a description of the NOCMPAT compile option.
- r is an integer expression that represents the relative position of a record within the data set associated with u.

The data set 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
```

After the FIND statement is executed, the value of IVAR is 50. After the READ statement is executed, the value is 51.

General Example—Direct-Access Operations

```
     DEFINE FILE 8(1000,72,L,ID8)
     DIMENSION A(100),B(100),C(100),D(100),E(100),F(100)
15   FORMAT (6F12.4)
     FIND (8'5)
50   ID8=1
     DO 100 I=1,100
100  READ (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
     •
     •
     DO 200 I=1,100
200  WRITE (8'ID8+4,15)A(I),B(I),C(I),D(I),E(I),F(I)
     •
     •
     END
```

The general example illustrates the ability of direct-access statements to gather and disperse data in an order you designated. The first DO loop in the example fills arrays A through F with data from the 5th, 10th, 15th,. . ., and 500th records of the data set identified by reference number 8. Array A receives the first value in every fifth record, B the second value, and so on, as specified by FORMAT statement 15 and the I/O list of the READ statement. At the end of the read operation, the records have been dispersed into arrays A through F. At the conclusion of the first DO loop, ID8 has a value of 501.

The second DO loop in the example groups the data items from each array as specified by the I/O list of the WRITE statement and FORMAT statement 15. Each group of data items is placed in the data set identified by reference number 8. Writing begins at the 505th record and continues at intervals of five until record 1000 is written, if ID8 is not changed between the last READ statement and the first WRITE statement.

Chapter 6. Data Initialization Statement

A DATA statement is a non-executable data initialization statement that is used to define initial values of variables, array elements, and arrays.

General Form of the DATA Statement

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

where:

- Each k is a list containing variables, array elements (in which case the subscript quantities must be unsigned integer constants), or array names. Dummy arguments may not appear in the list.
- Each d is a list of constants (integer, real, logical, hexadecimal, or literal), 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.

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, 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.

An initially defined variable, array, or array element may not be located in a blank common or global area. They may be defined in a labeled common area, but only in a BLOCK DATA subprogram.

This statement cannot precede a PROGRAM, SUBROUTINE, FUNCTION, or IMPLICIT statement. Otherwise, a DATA statement can appear anywhere in the program after any specification statements that refer to the initialized data item.

Examples:

```
DIMENSION D(5,10)
DATA A, B, C/5.0,6.1,7.3/,D,E/25*1.0,25*2.0,5.1/
```

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 25 elements of the array D are to be initialized to the value 1.0, the remaining 25 elements of D to the value 2.0, and the variable E to the value 5.1.

```
DIMENSION A(5), B(3,3)
DATA A/5*1.0/,B/9*2.0/,C/'FOUR'/
```

The DATA statement specifies that all the elements in the arrays A and B are to be initialized to the values 1.0 and 2.0, respectively. In addition, the variable C is to be initialized with the literal data constant 'FOUR'.

Chapter 7. Specification Statements

The specification statements are non-executable statements that provide the compiler with information about the nature of data used in the source program. In addition, they supply the information required to allocate locations in storage for this data. Specifications must precede statement function definitions, which must precede the program part containing at least one executable statement.

Type Statements

There are two kinds of type statements: the IMPLICIT specification statement; and the REAL, INTEGER, and LOGICAL explicit specification statements.

The IMPLICIT statement enables you to:

- Specify the type (including length) of all variables, arrays, and user-supplied functions whose names begin with a particular letter

The explicit specification statements enable you to:

- Specify the type (including length) of a variable, array, or user-supplied function of a particular name
- Specify the dimensions of an array

The explicit specification statements override the IMPLICIT statement, which, in turn, overrides the predefined convention for specifying type.

IMPLICIT Statement

General Form of the IMPLICIT Statement

IMPLICIT $type_1*s_1(a_{11},a_{12},...),$
$...,type_n*s_n(a_{n1},a_{n2},...)$

where:

- Type is INTEGER, REAL, or LOGICAL.
- Each *s is optional and represents one of the permissible length specifications for its associated type.
- Each 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 (for example, (A–D).

The IMPLICIT specification statement, if present, must be the first statement in a main program (the second if a PROGRAM statement is present) and the second statement in a subprogram. There can be only one IMPLICIT statement per program or subprogram. The IMPLICIT specification statement enables you to declare the type of the variables appearing in your program (integer or real) by specifying that variables beginning with certain designated letters are of a certain type. Furthermore, the IMPLICIT statement allows you to declare the number of bytes to be allocated for each in the group of specified variables. The types that a variable may assume, along with the permissible length specifications, are as follows:

Type	*Length*	*Specification (in bytes)*
INTEGER	2 or 4	(standard length is 4. See Appendix C for a description the NOCMPAT compile option)
REAL	4 or 8	(standard length is 4)
LOGICAL	4	

If the standard length specification is desired, the *s may be omitted. If the optional length specification is desired, the *s must be included within the IMPLICIT statement.

Examples:

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

All variables beginning with the characters A through H and O through $ are declared as INTEGER. Since no length specification is explicitly given (the *s was omitted), four bytes, the standard length for INTEGER, are allocated for each variable (see Appendix C).

All other variables (those beginning with the characters I through N) are declared as REAL with four bytes allocated for each.

Note that the statement in this example exactly reverses the predefined convention.

```
IMPLICIT INTEGER*2(A-H), REAL(I-K)
```

All variables beginning with the characters A through H are declared as integer with two bytes allocated for each. All variables beginning with the characters I through K are declared as real with four bytes allocated for each.

Since the remaining letters of the alphabet, L through Z (and $), are left undefined by the IMPLICIT statement, the predefined convention will remain in effect. Thus, variables beginning with the letters L, M, and N are integer, each with a standard length of four bytes (see Appendix C), and variables beginning with the letters O through $ are real, each with a standard length of four bytes.

Explicit Specification Statement

General Form of the Explicit Statement

$\texttt{type*s } a_1 * s_1(k_1)/x_1/,$
$a_2 * s_2(k_2)$
$/x_2/, \ldots, a_n * s_n(k_n)/x_n/$

where:

- type is INTEGER, REAL, or LOGICAL.
- Each *s is optional and represents one of the permissible length specifications for its associated type.
- Each a is a variable, array, or function name (see the section "Subprograms").
- Each k is optional and gives dimension information for arrays. Each k is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.
- Each /x/ is optional and represents initial data values. Dummy arguments may not be assigned initial values.

The explicit specification statements declare the type (INTEGER, REAL, 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 (that is, predefined convention and the IMPLICIT statement). In addition, the information necessary to allocate storage for arrays (dimension information) may be included within the statement.

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 literal or hexadecimal data may be used for any type. 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 explicit specification statement, the dimension information for the array must be in the explicit specification statement or in a preceding DIMENSION, GLOBAL, or COMMON statement. An initial data value may not be assigned to a function name, but a function name may appear in an explicit specification statement.

Initial data values cannot be assigned to variables or arrays in blank common or global. The BLOCK DATA subprogram must be used to assign initial values to variables and arrays in labeled common or global.

In the same manner in which the IMPLICIT statement overrides the predefined convention, the explicit specification statements override the IMPLICIT statement and predefined convention. If the length specification is omitted (that is, *s), the standard length per type is assumed.

Examples:

```
INTEGER*2 ITEM/76/, VALUE
```

This statement declares that the variables ITEM and VALUE are of type integer, each with two bytes of storage reserved. In addition, the variable ITEM is initialized to the value 76.

```
REAL BAKER, HOLD, VALUE, ITEM(5,5)
```

This statement declares that the variables BAKER, HOLD, VALUE, and the array named ITEM are of type real. In addition, it declares the size of the array ITEM. The variables BAKER, HOLD, and VALUE have four bytes of storage reserved; and the array named ITEM has 100 bytes of storage reserved.

```
REAL A(5,5)/20*6.9E2,5*1.0/,B(100)/100*0.0/
```

This statement declares the size of each array, A and B, and their type (real). The array A has 100 bytes of storage reserved (four for each element in the array) and the array B has 400 bytes of storage reserved (four for each element). In addition, the first 20 elements in the array A are initialized to the value 6.9E2 and the last five elements are initialized to the value 1.0. All 100 elements in the array B are initialized to the value 0.0.

```
INTEGER*2 AE, BE*4, CE(5)
```

This statement declares AE to be INTEGER*2, BE to be INTEGER*4, and CE to be an INTEGER*2 array consisting of five elements.

DIMENSION Statement

General Form of the DIMENSION Statement

DIMENSION $a_1(k_1), a_2(k_2), \ldots a_n(k_n)$

where:

- Each a is an array name.
- Each k is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. (It may also be provided by a type statement or a COMMON statement.) The following examples illustrate how this information may be declared.

Examples:

```
DIMENSION A(10),ARRAY (5,5,5), LIST(10,100)
DIMENSION B(25,25),TABLE (5,10,15)
```

The first statement defines three arrays: A, ARRAY, and LIST. The array A is a single dimension array consisting of ten elements, the array ARRAY is a three-dimensional array, and LIST is a two-dimensional array. The second statement defines a two-dimensional array, B, and a three-dimensional array, TABLE.

DOUBLE PRECISION Statement

General Form of the DOUBLE PRECISION Statement

DOUBLE PRECISION $a_1,(k_1),a_2(k_2),a_3(k_3),\ldots,a_n$

where:

- Each a represents a variable, array, or function name (see the section "Subprograms").
- Each k is optional and is composed of one through seven unsigned integer constants, separated by commas, that represent the maximum value of each subscript in the array.

The DOUBLE PRECISION statement explicitly specifies that each of the variables a is of type double-precision. This statement overrides any specification of a variable made by either the predefined convention or the IMPLICIT statement. The specification is identical to that of type REAL*8, but it cannot be used to define initial data values.

COMMON Statement

General Form of the COMMON Statement

COMMON/r_1/$a_{11}(k_{11}),a_{12}(k_{12}),\ldots$/
r_n/$a_{n1}(k_{n1})a_{n2}(k_{n2}),\ldots$

where:

- Each a is a variable name or array name that is not a dummy argument.
- Each k is optional and is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.
- Each 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_1 denotes blank common, the first two slashes are optional.

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 two purposes: to conserve storage, by avoiding more than one allocation of storage for variables and arrays used by several program units; and to implicitly transfer arguments between a calling program and a subprogram. Arguments passed in a common area do not appear in the argument lists of either the calling program or subprogram. Arguments in common are subject to the same rules with regard to type, length, etc., as arguments passed in an argument list. (These rules are described in the section dealing with subprograms.)

Since the entries in a common area share storage locations, the order in which they are entered is significant when the common area is used to transmit arguments.

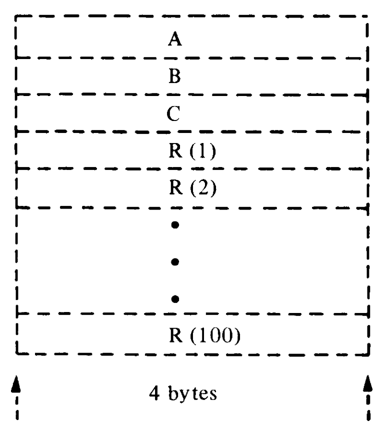
Examples:

Main Program	*Subprogram*

```
COMMON A, B, C, R(100)    SUBROUTINE MAPMY
REAL A,B,C                COMMON X, Y, Z, S(100)
INTEGER*4 R               REAL X,Y,Z
•                         INTEGER*4 S
•                         •
•                         •
CALL MAPMY                •
•                         •
•                         •
•                         •
```

The statement COMMON A,B,C,R(100) in the main program would cause 206 words of storage (four bytes per variable or array element) to be reserved in the following order:

Beginning of common area

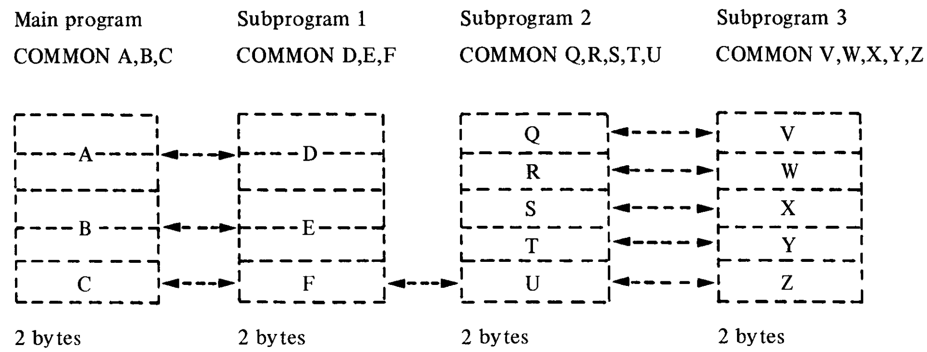


The statement COMMON X, Y, Z, S(100) in the subprogram would then cause the variables X, Y, Z, and S(1),...,S(100) to share the same storage space as A, B, C, and R(1),...,R(100), respectively.

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

Main Program:	COMMON	A,B,C	(where A and B have previously been declared as INTEGER*4, and C as INTEGER*2)
Subprogram 1:	COMMON	D,E,F	(where D and E have previously been declared as INTEGER*4, and F as INTEGER*2)
Subprogram 2:	COMMON	Q,R,S,T,U	(where all have previously been declared as INTEGER*2)
Subprogram 3:	COMMON	V,W,X,Y,Z	(where all have previously been declared as INTEGER*2)

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. However, the main program and subprogram 1 cannot, by assigning values to the variables A and B, or D and E, 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. Values can also be transmitted 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 and E destroys the values of S, T, X, and Y (and vice versa).

Blank and Labeled Common

In the preceding example, the common storage area (common block) is called a blank common area, that is, no particular name was given to that area of storage. The variables that appeared in the COMMON statement were assigned locations relative to the beginning of this 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 alphameric characters (the first of which is alphabetic); those blocks having the same name occupy the same storage space. This permits a calling program to share one common block with one subprogram and another common block with another subprogram and also facilitates program documentation.

The differences between blank and labeled common are:

- There is only one blank common in an executable program, and it has no user-assigned name; there may be many labeled commons, each with its own name.
- Each program unit which uses a given labeled common must define it to be of the same length; blank common may have different lengths in different program units.
- Variables and array elements in blank common cannot be assigned initial values; variables and array elements in labeled common may be assigned initial values by DATA statements or explicit specification statements only in a BLOCK DATA subprogram.

Those variables that are to be placed in labeled (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 is distinguished from labeled common by placing two consecutive slashes before the variables in blank common 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
```

Programming Considerations

1. There is no restriction as to the number of program units which may have COMMON statements, but a COMMON statement in only one program unit serves no purpose other than strictly ordering the arrangement of variables in common. It would normally have at least one counterpart in another program unit.
2. There may be more than one COMMON statement in a program unit. A variable or array name may not appear more than once in a COMMON statement, in more than one COMMON statement in the same program unit, or in both a COMMON and GLOBAL statement.
3. It may arise that not all program units need refer to all of the variables and arrays in common. Thus, in order to maintain correct positioning, so-called dummy variables can be inserted into the COMMON statement list. These dummy variables are not referenced anywhere else in the program unit. Their function is simply to allow you to position variable and array names that otherwise would be in the wrong locations in a COMMON statement.

   Example:

Main Program	*Subprogram*
`COMMON A, B, C, D`	`COMMON DUMMY1, BETA,DUMMY3,DELTA`

4. Because the main program and subprograms have access to common storage locations via the COMMON statement, they have, in effect, a way of communicating with each other. This means that a value computed in one program unit and placed in common storage can be used by another program unit in much the same manner as if it were passed as an argument. This idea will become clearer when CALL statements and function references are discussed in Chapter 8.

EQUIVALENCE Statement

General Form of the EQUIVALENCE Statement

EQUIVALENCE ($a_{11}, a_{12}, a_{13}, \ldots$),
($a_{21}, a_{22}, a_{23}, \ldots$), . . .

where:

- Each a is a variable or array element. It 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 (that is, 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 or logical expression (that is, its position relative to the first element of each dimension of the array). In either case, the subscripts themselves must be integer constants.

All the elements within a single set of parentheses share the same storage locations. The order of appearance of names within an equivalence group is immaterial.

The EQUIVALENCE statement provides the option for controlling the allocation of data storage within a single program unit. 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.

Since arrays are stored in a predetermined order as discussed previously, equivalencing two elements of two different arrays will implicitly equivalence other elements of the two arrays. (The one exception occurs when the first element of an array is equivalenced to the last element of another array.) 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.

Variables that appear in COMMON or GLOBAL statements cannot be equivalenced to each other. However, a variable can be made equivalent to a variable in a COMMON or GLOBAL statement in the same program unit. If a variable that is so equivalenced is an element of an array, the implicit equivalencing of the rest of the elements of the array can extend the size of common or global as shown below. But the size of common or global cannot be extended so that elements are added before the beginning of the established common or global area.

Examples:

Assume that in the initial part of a program, an array C of size 10x10 is needed; in the final stages of the program, C is no longer used, but arrays A and B of sizes 5x5 and 10, respectively, are used. The elements of all three arrays are of the type REAL*4. Storage space can then be saved by using the statements:

```
DIMENSION C(10,10), A(5,5), B(10)
EQUIVALENCE (C(1), A(1)), (C(26), B(1))
```

The array A, which has 25 elements, can occupy the same storage as the first 25 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 26 through 35 of array C.

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

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.

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

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:

```
GLOBAL 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)

Other Specification Statements

There are three other specification statements: EXTERNAL, PROGRAM, and GLOBAL. These statements are discussed in Chapter 8.

Chapter 8. Subprograms

It is sometimes desirable to write a program which, at various points, requires the same computation to be performed with different data for each calculation. It would simplify the writing of that program if the statements required to perform the desired computation could be written only once and then could be referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written with this object in mind. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The FORTRAN IV language provides for this situation through the use of subprograms. There are two classes of subprograms: FUNCTION subprograms and SUBROUTINE subprograms. In addition, there is a group of FORTRAN IV subprograms supplied by the Series/1 Mathematical and Functional Subroutine Library program product and the Series/1 FORTRAN IV Realtime Subroutine Library. See Appendix B for descriptions of these libraries. FUNCTION subprograms differ from SUBROUTINE subprograms in that FUNCTION subprograms return at least one value to the calling program, whereas SUBROUTINE subprograms need not return any. In addition, the method of referring to the two kinds of subprograms is different.

A subprogram must never refer to itself directly or indirectly.

Statement functions are also discussed in this section since they are similar to FUNCTION subprograms. The difference is that subprograms are not in the same program unit as the program unit referring to them, whereas statement function definitions and references are in the same program unit.

Naming Subprograms

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

Type Declaration of a Statement Function: Such declaration may be accomplished by the predefined convention, by the IMPLICIT statement, or by the explicit specification statements. Thus, the rules for declaring the type of variables apply to statement functions.

Type Declaration of FUNCTION Subprograms: The declaration 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. Note that if the predefined convention is not used, the function must be typed both in the function subprogram and in each program unit that refers to the function.

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

Functions

A function is a statement of the relationship between a number of variables. To use a function in FORTRAN IV, it is necessary to:

- Define the function (that is, specify which calculations are to be performed)
- Refer to the function by name where required in the program

Function Definition

There are three steps in the definition of a function in FORTRAN IV:

1. The function must be assigned a name by which it can be called.
2. The dummy arguments of the function must be stated.
3. The procedure for evaluating the function must be stated.

Items 2 and 3 are discussed in detail in the sections dealing with the specific subprogram, "Statement Functions" and "FUNCTION Subprograms".

Function Reference

When the name of a function, followed by a list of its arguments, appears in any FORTRAN IV expression, it refers to the function and causes the computations to be performed as indicated by the function definition. The resulting quantity (the function value) replaces the function reference in the expression and assumes the type of the function. The type of the name used for the reference must agree with the type of the name used in the definition.

Statement Functions

A statement function definition specifies operations to be performed whenever that statement function name appears as a function reference in another statement in the same program unit.

General Form of a Statement Function Definition

name($a_1, a_2, a_3, \ldots, a_n$)=expression

where:

- name is the statement function name. The name consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic.
- Each a is a dummy argument. It must be a distinct variable (that is, 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.

The dummy arguments (enclosed in parentheses) following the function name are dummy variables. The arguments given in the function reference are substituted for the dummy variables 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
```

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

The following are valid statement function definitions and statement function references:

Definition	*Reference*
`SUM(A,B,C,D)=A+B+C+D`	`NET=X-SUM(T1,T2,T3,T4)`
`FUNC(Z)=A+X*Y*Z`	`ANS=FUNC(RESULT)`

The following are invalid statement function definitions:

`SUBPRG(3,J,K)=3*I+J**3`	(Arguments must be variables)
`SOMEF(A(I),B)=A(I)/B+3.`	(Arguments must not be array elements)
`SUBPROGRAM(A,B)=A**2+B**2`	(Function name exceeds limit of six characters)
`3FUNC(D)=3.14*E`	(Function name must begin with an alphabetic character)
`ASF(A)=A+B(I)`	(Expression may not contain an array element)
`BAD(A,B)=A+B+BAD(C,D)`	(Definition not permitted to refer to itself)
`NOGOOD(A,A)=A*A`	(Arguments are not distinct variable names)

The following are invalid statement function references (the functions are defined as above):

`WRONG=SUM(COUNT1,COUNT2)`	(Number of arguments does not agree with above definition)
`MIX=FUNC(I)`	(Type of argument does not agree with above definition)

FUNCTION Subprograms

The FUNCTION subprogram is a FORTRAN IV subprogram consisting of a FUNCTION statement followed by other statements including a RETURN statement and an END statement. It is an independently written program that is executed wherever its name is referred to in another program.

General Form of the FUNCTION Statement

type FUNCTION name*s $(a_1, a_2, a_3, \ldots, a_n)$

where:

- type is INTEGER, REAL, DOUBLE PRECISION, or LOGICAL. Its inclusion is optional.
- name is the name of the FUNCTION, which consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic.

- s represents one of the permissible length specifications for its associated type. It may be included optionally only when type is specified.
- Each a is a dummy argument. It must be a distinct variable or array name (that is, it may appear only once within the statement) or a 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.

The FUNCTION statement must be the first statement in the subprogram. The FUNCTION subprogram may contain any FORTRAN IV statement except a PROGRAM statement, another FUNCTION statement, a SUBROUTINE statement, a BLOCK DATA statement, or a DEFINE FILE statement. If an IMPLICIT statement is used in a FUNCTION subprogram, it must immediately follow the FUNCTION statement.

The name of the function, or one of the entry names (see "Multiple Entry into a Subprogram " in this chapter), must be assigned a value at least once in the subprogram—as the variable name to the left of the equal sign in an arithmetic or logical assignment statement, as an argument of a CALL statement or external function reference that is assigned a value by the subroutine or function referred to, or in the list of a READ statement within the subprogram.

The FUNCTION subprogram may also use one or more of its arguments or any quantity in common or global to return a value to the calling program.

The dummy arguments of the FUNCTION subprogram (that is, $a_1,a_2,a_3,\ldots,a_n$) may be considered to be dummy names. These are associated at the time of execution by the actual arguments supplied in the function reference in the calling program. Additional information about arguments is in the section "Dummy Arguments in a FUNCTION or SUBROUTINE Subprogram".

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the FUNCTION subprogram is illustrated in the following example:

Calling Program	*FUNCTION Subprogram*
•	`FUNCTION CALC (A,B,J)`
•	•
•	•
`ANS=ROOT1*CALC(X,Y,I)`	•
•	`I=J*2`
•	•
•	•
•	•
	`CALC=A**I/B`
	`RETURN`
	`END`

In this example, the values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The value of CALC is computed, and this value is returned to the calling program, where the value of ANS is computed. The variable I in the argument list of CALC in the calling program is not the same as the variable I appearing in the subprogram.

Calling Program	*FUNCTION Subprogram*
`INTEGER*2 CALC`	`INTEGER FUNCTION CALC*2(I,J,K)`
•	•
•	•
•	•
`ANS=ROOT1*CALC(N,M,L)`	`CALC=I+J+K**2`
•	•
•	•
•	•
	`RETURN`
	`END`

In this example, the FUNCTION subprogram CALC is declared as type INTEGER of length 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 a RETURN statement and 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. Unlike the FUNCTION subprogram, it need not return any results to the calling program. The SUBROUTINE subprogram is referenced by the CALL statement.

General Form of the SUBROUTINE Statement

SUBROUTINE name ($a_1,a_2,a_3,\ldots,a_n$)

where:

- name is the SUBROUTINE name, which consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic.
- Each a is a distinct dummy argument (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 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. See the section "Dummy Arguments in Subprograms".

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

The SUBROUTINE statement must be the first statement in the subprogram. The SUBROUTINE subprogram may contain any FORTRAN IV statement except a PROGRAM statement, another SUBROUTINE statement, a FUNCTION statement, a BLOCK DATA statement, or a DEFINE FILE statement.

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. An argument so used will appear on the left side of an arithmetic or logical assignment statement, in the list of a READ statement within the subprogram, or as an argument in a CALL statement or function reference that is assigned a value by the subroutine referred to. The subroutine name must not appear in any other statement in the SUBROUTINE subprogram.

The dummy arguments (a_1, a_2, a_3,...,a_n) may be considered dummy names that are associated at the time of execution with the actual arguments supplied in the CALL statement. Additional information about dummy arguments is in the section dealing with dummy arguments in a FUNCTION or SUBROUTINE subprogram.

CALL Statement

General Form of the CALL Statement

CALL name (a_1,a_2,a_3,...,a_n)

where:

- name is the name of a SUBROUTINE subprogram or an ENTRY name in the subprogram.
- Each a is an actual argument that is being supplied to the SUBROUTINE subprogram. The argument may be a variable name, array element name, array name, literal, arithmetic or logical expression, or subprogram name. (Subprogram names passed as arguments must be specified in an EXTERNAL statement.)
- a may also be of the form & n, where n is a statement number of an executable statement in the calling program to which control may be returned. & indicates that n is a statement number instead of an integer constant.

The CALL statement is used to call a SUBROUTINE subprogram.

Examples:

```
CALL OUT
CALL MATMPY (X,5,40,Y,7,2)
CALL QDRTIC (X,Y,Z,ROOT1,ROOT2)
CALL SUB1(X+Y*5,ABDF,SINE)
CALL SUB(A,B,&50,&70,&85)
```

The CALL statement transfers control to the SUBROUTINE subprogram and associates the dummy variables with the value of the actual arguments that appear in the CALL statement.

Calling Program	*SUBROUTINE Subprogram*
DIMENSION X(100),Y(100)	
•	SUBROUTINE COPY(A,B,N)
•	DIMENSION A(100),B(100)
•	DO 10 I=1,N
CALL COPY (X,Y,100)	10 B(I)=A(I)
•	RETURN
•	END
•	

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the SUBROUTINE subprogram is illustrated.

Subroutine COPY copies array A into array B within the subprogram. In this particular call, the subroutine arrays A and B are associated with the calling program arrays X and Y, respectively, and the variable N in the subroutine is associated with the value 100. Thus, a call to subroutine COPY in this instance results in the 100 elements of array X being copied into the 100 elements of array Y.

RETURN and END Statements in Subprograms

All FUNCTION and SUBROUTINE subprograms must contain an END statement and at least one RETURN statement. The END statement specifies the physical end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns the computed function value and control to the calling program. (In a main program, a RETURN statement serves the same function as a STOP statement.)

The form of the END statement is given above under "Control Statements".

General Form of the RETURN Statement

```
RETURN
RETURN   i
```

where:

- i is an integer constant or variable of length 4 (see Appendix C for a description of the NOCMPAT compile option) whose value (n, for example,) denotes the nth statement number in the argument list of a SUBROUTINE statement; i may be specified only in a SUBROUTINE subprogram.

Examples:

```
     FUNCTION DAV (D,E,F)
     IF (D-E) 10, 20, 30
10   A=D+2.0*E
     •
     •
     •
 5   A=F+2.0*E
     •
     •
     •
20   DAV=A+D**2
     •
     •
     •
     RETURN
30   DAV=D**2
     •
     •
     •
     RETURN
     END
```

If the result of (D-E) is negative or zero, the first RETURN statement will be executed. If the result is positive, the second RETURN will be executed.

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. Returns of the type RETURN may be made in either a SUBROUTINE or FUNCTION subprogram (see "RETURN and END Statements in Subprograms"). Returns of the type RETURN i may only be made in a SUBROUTINE subprogram. The value of i must be within the range of the argument list.

Calling Program	*Subprogram*
•	SUBROUTINE SUB (X,Y,Z,*,*)
•	•
•	•
10 CALL SUB (A,B,C,&30,&40)	•
20 Y=A+B	100 IF (M) 200,300,400
•	200 RETURN
•	300 RETURN 1
•	400 RETURN 2
30 Y=A+C	END
•	
•	
•	
40 Y=B+C	
•	
•	
•	
END	

In the preceding example, execution of statement 10 in the calling program causes entry into subprogram SUB. When statement 100 is executed, the return to the calling program will be to statement 20, 30, or 40, if M is less than, equal to, or greater than zero, respectively.

A CALL statement that uses a RETURN i form may be best understood by comparing it to a CALL and computed GO TO statement in sequence. For example, the following CALL statement:

```
CALL SUB (P,&20,Q,&35,R,&22)
```

is equivalent to:

```
CALL SUB (P,Q,R,I)
GO TO (20,35,22),I
```

where:

- the index I is assigned a value of 1, 2, or 3 in the called subprogram.

Dummy Arguments in Subprograms

The dummy arguments of a subprogram appear after the FUNCTION or SUBROUTINE name and are enclosed in parentheses. They are associated at the time of execution with the actual arguments supplied in the CALL statement or function reference in the calling program. The dummy arguments must correspond in number, order, and type to the actual arguments. For example, if an actual argument is an integer constant, then the corresponding dummy argument must be an integer variable of the same length. If a dummy argument is an array, the corresponding actual argument must be (1) an array or (2) an array element. In the first instance, the size of the dummy array must not exceed the size of the actual array. In the second, the size of the dummy array must not exceed the size of that portion of the actual array which follows and includes the designated element.

The actual arguments can be:

- A literal, arithmetic, or logical constant
- Any type of variable or array element
- Any type of array name
- Any type of arithmetic or logical expression
- The name of a FUNCTION or SUBROUTINE subprogram or an entry name in a subprogram
- A statement number (for a SUBROUTINE subprogram using RETURN i).

An actual argument which is the name of a subprogram or entry must be identified by an EXTERNAL statement in the calling program unit containing that name. Hexadecimal constants cannot be actual arguments. If a literal is passed to a function or subroutine, the argument passed is the literal as defined, without delimiting apostrophes or the preceding wH specification.

A dummy argument is an array when an appropriate DIMENSION or explicit specification statement appears in the subprogram. None of the dummy arguments may appear in an EQUIVALENCE, COMMON, or GLOBAL statement.

The subprogram may use one or more of its arguments or any quantity in common or global to return a value to the calling program.

If a dummy argument is assigned a value in a subprogram, the corresponding actual argument must be a variable, an array element, or an array. A constant or expression should not be written as an actual argument unless you are certain that the corresponding dummy argument is not assigned a value in the subprogram.

A referenced subprogram cannot assign new values to dummy arguments which are associated with other dummy arguments within the subprogram or with variables in common or global areas. For example, if the subroutine DERIV is defined as

```
SUBROUTINE DERIV (X,Y,Z)
COMMON W
```

and if the statements

```
COMMON B
•
•
•
CALL DERIV (A,B,A)
```

are included in the calling program, then X, Y, Z, and W cannot be assigned new values by the subroutine DERIV. Dummy arguments X and Z cannot be defined because they are both associated with the same argument, A; dummy argument Y cannot because it is associated with an argument, B, which is in COMMON; and the variable W cannot because it is also associated with B.

Multiple Entry into a Subprogram

The standard 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.

General Form of the ENTRY Statement

ENTRY name $(a_1,a_2,a_3,...a_n)$

where:

- name is the name of an entry point (see the section "Naming Subprograms").
- Each a is a dummy argument corresponding to an actual argument in a CALL statement or in a function reference (see the section "Dummy Arguments in Subprograms"). In functions, each array name must be of the same type as the function name.

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.

The dummy arguments in the ENTRY statement need not agree in order, type, or number with the dummy arguments in the SUBROUTINE or FUNCTION statement 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 arguments 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.

All arguments in Series/1 FORTRAN IV are passed by name. This means that when an argument is accessed, its current value in the calling program is used. Similarly, when an argument is changed, the current value in the calling program is also immediately changed.

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 (that is, the function name and entry names) must be the same; the variables are treated as if they were equivalenced.

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.

Examples:

Calling Program

```
     REAL TABLE(100), W(100)
     •
     •
     •
     TABLE(1)=FUNC(W(1),X,Y,Z)
     DO 5 I=2,100
     TABLE(I)=ENT(W(I))
     •
     •
     •
5    CONTINUE
     •
     •
     •
     END
```

Subprogram

```
FUNCTION FUNC(T,A,B,C)
•
•
•
ENTRY ENT(T)
•
•
•
FUNC=A * B+C ** T
RETURN
•
•
•
```

The FUNCTION subprogram is entered once at entry point FUNC and the dummy arguments become associated with the corresponding actual arguments. Thereafter, the FUNCTION subprogram is entered at entry point ENT, and only T becomes associated with the new actual argument.

Each time, the result of the FUNCTION subprogram is returned to the main program function by the variable FUNC.

Calling Program	*Subprogram*

Calling Program

```
     •
     •
     •
     CALL SUB1 (A,B,C,D,E,F)
     •
     •
     •
     CALL SUB2 (G,&10,&20)
     Y=G
     •
     •
     •
     CALL SUB3 (&10,&20)
     Y=A+B
     •
     •
     •
10   Y=C+Dy
     •
     •
     •
20   Y=E+F
     •
     •
     •
```

Subprogram

```
SUBROUTINE SUB1 (U,V,W,X,Y,Z)

RETURN
ENTRY SUB2 (T,*,*)
U=V* W+T
ENTRY SUB3 (*,*)
X=Y**Z
50  IF (W) 100, 200, 300
100 RETURN 1
200 RETURN 2
300 RETURN
END
```

In this example, a call to SUB1 merely performs initialization. A subsequent call to SUB2 or SUB3 causes execution of a different section of the SUB1 subroutine. Then, depending upon the result of the arithmetic IF statement at statement 50, control returns to the calling program at statement 10, 20, or the statement following the call.

EXTERNAL Statement

General Form of the EXTERNAL Statement

EXTERNAL $a_1, a_2, a_3, \ldots, a_n$

where:

- Each a is a name of a subprogram that is passed as an argument to other subprograms.

The EXTERNAL statement is a specification statement, and must precede statement function definitions and all executable statements.

If the name of a FORTRAN IV-supplied intrinsic function is used in an EXTERNAL statement, the function is not used from the FORTRAN IV-supplied library when it appears as a function reference. Instead, it is assumed that the function is supplied by the user.

The name of any subprogram that is passed as an argument to another subprogram must appear in an EXTERNAL statement in the calling program. For example, assume that SUB and MULT are subprograms names in the following statements:

Calling Program	*Subprogram*
`EXTERNAL MULT`	`SUBROUTINE SUB(K,M,Z)`

```
•                                 IF (K) 4,6,6
•                                 4    D=M(K,Z**2)
•                                 •
CALL SUB(J,MULT,C)                •
•                                 •
•                                 6    RETURN
•                                 END
```

In this example, the function name MULT is used as an argument in the subroutine SUB. The function name MULT is passed to the dummy variable M and the variables J and C are passed to the dummy variables K and Z, respectively. The function MULT is executed only if the value of K is negative.

Calling Program	*Subprogram*

```
•                                 SUBROUTINE SUB(W,X,M,N)
•                                 •
•                                 •
CALL SUB(A,B,MULT(C,D),37)        •
•                                 RETURN
•                                 END
•
```

In this example, an EXTERNAL statement is not required because the function MULT is not an argument; it is executed first and the result becomes the argument.

BLOCK DATA Subprograms

To initialize variables in a labeled (named) common block, a separate subprogram must be written. This separate subprogram contains only the BLOCK DATA, DATA, COMMON, DIMENSION, EQUIVALENCE, and type statements associated with the data being defined, as well as the END statement. Data may not be initialized in unlabeled common or global, labelled or unlabelled.

General Form of the BLOCK DATA Statement

```
BLOCK DATA
```

- The BLOCK DATA subprogram may not contain any executable statements, statement function definitions, or FORMAT, GLOBAL, DEFINE FILE, PROGRAM, FUNCTION, SUBROUTINE, ENTRY, or DEBUG statements.
- The BLOCK DATA statement must be the first statement in the subprogram. If an IMPLICIT statement is used in a BLOCK DATA subprogram, it must immediately follow the BLOCK DATA statement. Statements which provide initial values for data items cannot precede the COMMON statements which define those data items.
- Any main program or subprogram using a common block must contain a COMMON statement defining that block. If initial values are to be assigned, a BLOCK DATA subprogram is necessary.
- All elements of a common block must be listed in the COMMON statement, even though they are not all initialized.
- Data may be entered into more than one common block in a single BLOCK DATA subprogram.
- Only one BLOCK DATA subprogram may be used to enter data into a particular common block.
- The BLOCK DATA subprogram must end with an END statement.

Inter-Program Communication

Series/1 FORTRAN IV allows you to construct applications or jobs that consist of multiple main programs executing. An application can contain one or more sequential steps. Each step executes a task set in a Realtime Programming System partition. A task set comprises one or more asynchronous tasks. A task is a single thread of execution through a unit of code. A FORTRAN IV task is a main program and its subprograms.

The Series/1 FORTRAN IV Realtime Subroutine Library provides a set of routines that can be called by FORTRAN IV programs to implement communications between asynchronously executing tasks and task sets. These routines are listed in Appendix B under "System Service Interface Subroutines". In addition, Series/1 FORTRAN IV provides the PROGRAM, INVOKE, and GLOBAL statements to aid in building multitask and multitask set applications.

PROGRAM Statement

General Form of the PROGRAM Statement

```
PROGRAM name
```

where:

- name is the name assigned to the main program. It consists of from one to six alphabetic or numeric characters, the first of which must be alphabetic.

A PROGRAM statement, if it appears, must be the first statement in a main program. It may not appear in a subprogram. If no PROGRAM statement appears in a main program, the program's name becomes MAIN by default. There is no type associated with a program name.

INVOKE Statement

General Form of the INVOKE Statement

```
INVOKE name
```

where:

- name is the 1 to 6-character task set.

Execution of the INVOKE statement causes the named program to overlay the invoking program and receive control. The INVOKE statement may appear only in a main program. See the *Series/1 FORTRAN IV: User's Guide* for additional information.

A program which is invoked begins execution at its first executable instruction, that is, at the first executable statement of the main program.

GLOBAL Statement

The GLOBAL statement is used to define a storage area in much the same manner as a COMMON statement. The difference is that GLOBAL is used to communicate or share values among routines which may be executed asynchronously and/or between main programs which are invoked.

General Form of the GLOBAL Statement

$\mathtt{GLOBAL}\ /r_1/a_{11}(k_{11}),a_{12}(k_{12}),\ldots$
$/r_n/a_{n1}(k_{n1}),a_{n2}(k_{n2}),\ldots$

where:

- Each a is a variable name or array name that is a dummy argument.
- Each k is optional and is composed of one to seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array.

- Each r represents an optional global name consisting of one to 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 global. If r_1 denotes blank global, the first two slashes are optional.

The GLOBAL statement may appear in a main program or a subprogram. A program unit may contain any number of GLOBAL statements. All variables and arrays in these statements are strung together in the order of their appearance. A variable or array name may not appear more than once in a GLOBAL statement, in more than one GLOBAL statement, or in both a GLOBAL and a COMMON statement.

The global data area is for inter-program communication, although a main program may share a global data area with a subprogram. The local COMMON statement may still be used for this intra-program communication.

The global data area may be either blank or labeled, exactly as a common data area; however, variables in a global area may not be initialized in a BLOCK DATA subprogram.

Rules regarding the use of EQUIVALENCE are the same for a global data area as for a common data area.

Appendix A. Source Program Characters

Alphabetic Characters	*Numeric Characters*
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	
L	
M	
N	*Special Characters*
O	
P	(blank)
Q	+
R	-
S	/
T	=
U	.
V	)
W	*
X	,
Y	(
Z	' (apostrophe)
$	&

The 49 characters listed above constitute the set of characters acceptable by FORTRAN IV, except in literal data, where any valid card code is acceptable.

Appendix B. FORTRAN IV-Supplied and Optional Procedures

Series/1 FORTRAN IV, in conjunction with the Series/1 Mathematical and Functional Subroutine Library: Program Product 5719-LM1, and the Series/1 FORTRAN IV Realtime Subroutine Library: Program Product 5719-FO3, supplies you with the following types of procedures:

- Mathematical functions
- Service subroutines
- Bit interrogator functions
- Address constant function
- System service interface subroutines
- Subroutines which conform to the Instrument Society of America specifications ISA-S61.1-1976
  - Executive function
  - Process I/O
  - Time and Date
  - Bit interrogator and manipulator functions

FORTRAN IV procedures may be "in-line" or "out-of-line". In-line procedures are those functions which are inserted by the FORTRAN IV compiler at any point in the program where the function is referenced. An out-of-line procedure is located in a library, and the compiler generates an external reference to it. All subroutines are out-of-line procedures.

This appendix lists all procedures provided by FORTRAN IV, the Mathematical and Functional Subroutine Library (MFSL) and the FORTRAN IV Realtime Subroutine Library. The lists of procedures are arranged in two sections:

- Section I lists mathematical functions, service subroutines, the address constant function, and all bit interrogator/manipulator functions, including those bit manipulation functions specified in the Instrument Society of America (ISA) S61.1 standard. The procedures listed in Section I form the basic set of procedures supplied by FORTRAN IV and the MFSL. Detailed descriptions of the out-of-line mathematical functions and the FCTST, DVCHK, and OVERFL service subroutines are given in the MFSL publications. Detailed descriptions of all other out-of-line procedures are given in the *Series/1 FORTRAN IV: User's Guide*.
- Section II lists those procedures provided in the optional FORTRAN IV Realtime Subroutine Library. These procedures include the executive function, process I/O, system service interface, time, and date subroutines. Detailed descriptions of these procedures are given in the *Series/1 FORTRAN IV: User's Guide*.

Section I: Basic Procedures

Mathematical Functions

The mathematical functions are described in Figure B-1, parts I and II.

General function	Entry name	Definition	Arguments: No.	Arguments: Type²	Arguments: Range¹,²	Function value returned: Type²	Function value returned: Range¹,²	In line (I) Out of line (O)
Absolute value	IABS	$y = \lvert x \rvert$	1	INTEGER	Any INTEGER argument	INTEGER		I
	ABS		1	REAL *4	Any REAL argument	REAL *4		
	DABS		1	REAL *8		REAL *8		
Fix	IFIX	Convert from REAL to INTEGER	1	REAL *4	Any REAL argument	INTEGER		I
Float	FLOAT	Convert from INTEGER to REAL	1	INTEGER	Any INTEGER argument	REAL *4		I
	DFLOAT		1	INTEGER		REAL *8		
Maximum and minimum values	MAX0	$y = \max(x_1,...,x_n)$	⩾ 2	INTEGER	Any INTEGER argument	INTEGER		O
	AMAX0		⩾ 2	INTEGER		REAL *4		
	MAX1		⩾ 2	REAL *4	Any REAL argument	INTEGER		
	AMAX1		⩾ 2	REAL *4		REAL *4		
	DMAX1		⩾ 2	REAL *8		REAL *8		
	MIN0	$y = \min(x_1,...,x_n)$	⩾ 2	INTEGER	Any INTEGER argument	INTEGER		
	AMIN0		⩾ 2	INTEGER		REAL *4		
	MIN1		⩾ 2	REAL *4	Any REAL argument	INTEGER		
	AMIN1		⩾ 2	REAL *4		REAL *4		
	DMIN1		⩾ 2	REAL *8		REAL *8		
Modulo arithmetic	MOD	y - remainder $\left(\frac{x_1}{x_2}\right)$, i.e., $y = x_1$ (modulo x_2)	2	INTEGER	$x_2 \neq 0$	INTEGER		O
	AMOD		2	REAL *4		REAL *4		
	DMOD		2	REAL *8		REAL *8		
Obtain most significant part of a REAL argument	SNGL		1	REAL *8	Any REAL argument	REAL *4		I
Positive difference	IDIM	$y = x_1 - \min(x_1,x_2)$	2	INTEGER	Any INTEGER argument	INTEGER		O
	DIM		2	REAL *4	Any REAL argument	REAL *4		
Precision increase	DBLE		1	REAL *4	Any REAL argument	REAL *8		I
Transfer of sign	ISIGN	y - (sign x_2) · x_1, $x_1 \neq 0$	2	INTEGER	Any INTEGER argument	INTEGER		O
	SIGN		2	REAL *4	Any REAL argument	REAL *4		
	DSIGN		2	REAL *8		REAL *8		
Truncation	AINT	y = (sign x) · n where *n* is the largest integer ⩽ x	1	REAL *4	Any REAL argument	REAL *4		I
	INT		1	REAL *4		INTEGER		
	IDINT		1	REAL *8		INTEGER		

Notes.

1. $\gamma = 16^{63} \cdot (1^{16\,-6})$ for single precision and $16^{63} \cdot (1-16^{-14})$ for double precision.
2. If type or range is INTEGER and your program has been compiled in the CMPAT mode, the type/range must be INTEGER *4; if in the NOCMPAT mode, the type/range must be INTEGER *2.

Figure B-1. (Part 1 of 2) Mathematical functions

General function	*Entry name*	*Definition*	*Arguments*			*Function value returned*		*In line (I) Out of line (O)*
			No.	*Type*	*Range*[1]	*Type*	*Range*[1]	
Arctangent	ATAN	$y = \arctan x$	1	REAL *4	Any REAL argument	REAL *4 (in radians)	$-\frac{\pi}{2} \leq y \leq \frac{\pi}{2}$	O
	DATAN		1	REAL *8		REAL *8 (in radians)		
	ATAN2	$y = \arctan \frac{x_1}{x_2}$	2	REAL *4	Any REAL arguments except (0,0)	REAL *4 (in radians)	$-\pi < y \leq \pi$	
	DATAN2		2	REAL *8		REAL *8 (in radians)		
Exponential	EXP	$y = e^x$	1	REAL *4	$-180.218 \leq x \leq 174.673$	REAL *4	$0 \leq y \leq \gamma$	O
	DEXP		1	REAL *8		REAL *8		
Hyperbolic tangent	TANH	$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	1	REAL *4	Any REAL argument	REAL *4	$-1 \leq y \leq 1$	O
	DTANH		1	REAL *8		REAL *8		
Natural and common logarithm		$y = \log_e x$ or $y = \ln x$	1		$x > 0$			O
	ALOG		1	REAL *4		REAL *4	$-180.218 \leq y \leq 174.673$	
	DLOG		1	REAL *8		REAL *8		
		$y = \log_{10} x$	1		$x > 0$			
	ALOG10		1	REAL *4		REAL *4	$-78.268 \leq y \leq 75.859$	
	DLOG10		1	REAL *8		REAL *8		
Sine and cosine	SIN	$y = \sin x$	1	REAL *4 (in radians)	$\lvert x \rvert < (2^{18} \cdot \pi)$	REAL *4	$-1 \leq y \leq 1$	O
	DSIN		1	REAL *8 (in radians)	$\lvert x \rvert < (2^{50} \cdot \pi)$	REAL *8		
	COS	$y = \cos x$	1	REAL *4 (in radians)	$\lvert x \rvert < (2^{18} \cdot \pi)$	REAL *4	$-1 \leq y \leq 1$	
	DCOS		1	REAL *8 (in radians)	$\lvert x \rvert < (2^{50} \cdot \pi)$	REAL *8		
Square root	SQRT	$y = \sqrt{x}$ or $y = x^{1/2}$	1	REAL *4	$x \geq 0$	REAL *4	$0 \leq y \leq \gamma^{1/2}$	O
	DSQRT		1	REAL *8		REAL *8		

Notes.

1. $\gamma = 16^{63} \cdot (1-16^{-6})$ for single precision and $16^{63} \cdot (1-16^{-14})$ for double precision.

Figure B-1. (Part 2 of 2) Mathematical functions

Service Subroutines

Six service subroutines are available via the CALL statement: OVERFL, DVCHK, ERRXIT, FCTST, CLOSE, and EXIT. OVERFL, DVCHK, and FCTST are provided by the MFSL. ERRXIT, CLOSE, and EXIT are provided by FORTRAN IV.

1. *OVERFL:* This subroutine tests to determine if an overflow or underflow exception has occurred during execution. The source language statement is:

   ```
   CALL OVERFL ( i )
   ```

   where:

   - i is an integer variable set by the subroutine
   - i=1 indicates a floating-point overflow condition has occurred
   - i=2 indicates neither a floating-point overflow nor underflow condition has occurred
   - i=3 indicates a floating-point underflow condition has occurred

   If more than one such condition has occurred, the last one takes precedence. After execution of the call, the indicator is reset to 2.
2. *DVCHK:* This subroutine tests to determine if a floating-point divide-check exception has occurred during execution. The source language statement is:

   ```
   CALL DVCHK ( i )
   ```

where:

- i is an integer set by the subroutine
- i=1 indicates a divide-check exception has occurred
- i=2 indicates no divide-check exception has occurred

After execution of the CALL, the indicator is reset to 2.

3. *ERRXIT:* This subroutine is used to supply the name of the user-written subroutine to be given control if an error occurs during execution of an I/O statement. The source language statement is:

```
CALL ERRXIT (sub)
```

where:

- sub is the name of the user-written subroutine. This subroutine must also be declared in an EXTERNAL statement. The subroutine must have one INTEGER*2 dummy argument, which will be given the number of the error condition when FORTRAN IV passes it control. If the argument is zero, the effect of previous calls to ERRXIT is negated.

4. *FCTST:* This subroutine is used to determine if an illegal argument has been passed to a FORTRAN IV-supplied mathematical function, or invalid data has been read with F, E, D, I, or Z FORMAT codes. The source language statement is:

```
CALL FCTST (j,k)
```

where:

- j and k are integers set by the subroutine
- j=1 indicates one or more errors has occurred
- j=2 indicates no errors have occurred
- k= the value of the error code or codes

After execution of the CALL, the indicator j is reset to 2.

5. *CLOSE:* This subroutine is used to close a data set. The source language statement is:

```
CALL CLOSE (i)
```

where:

- i is an integer constant or variable set that you set to indicate the data set reference number of the data set you want to close.

6. *EXIT:* A call to the subroutine has essentially the same effect as the STOP statement. Both statements halt execution, but a CALL EXIT statement is not recognized as a logical end of the program by the compiler as is the STOP statement.

Bit Manipulator and Interrogator Functions

Figure B-2 describes the bit manipulator and interrogator functions. These functions allow access to individual bits within variables. They produce INTEGER*2 results and require INTEGER*2 arguments. (They are available only when a program is compiled with the NOCMPAT option.) In-line functions are supplied by FORTRAN IV. Out-of-line functions are provided by the MFSL.

Address Constant (ADCON) Function

This function produces the value of the location of the argument in main storage at execution time—its absolute address. This function is available only when a program is compiled with the NOCMPAT option. The function is of the form:

```
IADDR (arg)
```

where:

- arg is the entity whose address is desired

IADDR is compiled as an in-line function.

Entry name	Definition	Arguments No.	Arguments Type	Function value returned Type	In line (I) Out of line (O)	Bit value arg. 1	Bit value arg. 2	Bit value result
IOR	Value of inclusive OR of arg1 and arg2	2	INTEGER *2	INTEGER *2	I	0	0	0
						0	1	1
						1	0	1
						1	1	1
IEOR	Value of exclusive OR of arg1 and arg2	2	INTEGER *2	INTEGER *2	I	0	0	0
						0	1	1
						1	0	1
						1	1	0
IAND	Value of logical AND for arg1 and arg2	2	INTEGER *2	INTEGER *2	O	0	0	0
						0	1	0
						1	0	0
						1	1	1
ICOMP or NOT	Value of logical complement of arg	1	INTEGER *2	INTEGER *2	I	0	-	1
						1	-	0
ISHFT	Shifts arg1 by the count and direction of arg2; arg2 < 0 means shift right, arg2 > 0 means shift left, arg2=0 means no shift	2	INTEGER *2	INTEGER *2	O	Not applicable		
BTEST	If IAND (arg1,2^{arg2}) = 0, result is false. Else result is true.	2	INTEGER *2	INTEGER *2	O	Not applicable		
IBSET	Result = IOR (arg1, 2^{arg2})	2	INTEGER *2	INTEGER *2	O	Not applicable		
IBCLR	Result = IAND (arg1, NOT(2^{arg2})	2	INTEGER *2	INTEGER *2	O	Not applicable		

Figure B-2. Bit manipulator and interrogator functions

Section II: Series/1 FORTRAN IV Realtime Subroutine Library

The Series/1 FORTRAN IV Realtime Subroutine Library program product provides realtime system support for you. The procedures contained in this library are available only when a program is compiled with the NOCMPAT option. They require additional system support; therefore, only a summary of these procedures is available in this publication. See the *Series/1 FORTRAN IV: User's Guide* for information on using the FORTRAN IV Realtime Subroutine Library.

Date and Time Information

Obtain Time of Day

This subroutine allows a program to determine the current time of day. The form of this call is:

```
CALL TIME (j)
```

where:

- j designates an integer array into whose first three elements the absolute time of day will be placed. The contents of these elements shall be as follows:
  - First Element—Hours 0 to 23
  - Second Element—Minutes 0 to 59
  - Third Element—Seconds 0 to 59

Obtain Date

This subroutine allows a program to determine the current calendar date. The form of this call is:

```
CALL DATE (j)
```

where:

- j designates an integer array into whose first three elements the date will be placed. The contents of these elements shall be as follows:
  - First Element—AD year since zero
  - Second Element—Month 1 to 12
  - Third Element—Day 1 to 31

Executive Function Subroutines

The executive function subroutines provide you with the ability to start, stop, or delay the execution of programs. The calling sequences of the three executive function subroutines are similar in format and, therefore, they have been summarized as follows:

START	–	Requests execution of a specified program either immediately or after a time delay
TRNON	–	Requests execution of a specified program at a specified time of day
WAIT	–	Permits a program to suspend its own execution, gives control to the system, and resumes execution after a specified time delay

The format of the calling sequence is:

```
CALL START (i,j,k,m)
CALL TRNON (i,j,m)
CALL WAIT (j,k,m)
```

where:

- i specifies the name of the program to be executed
- j for START and WAIT specifies length of time, in units as specified by k, to delay before beginning or continuing execution of a program
- j for TRNON requests time of day designated by an array whose first three elements specify the hour, minute, and second
- k specifies units of time as either basic system clock counts, milliseconds, seconds, or minutes
- m return code indicating either a successful call or an error condition

Process Input and Output Subroutines

The process I/O subroutines allow you to access analog and digital points for both input and output. These subroutines return control to the calling program after the requested I/O operation is completed. The calling sequences of the process I/O subroutines are similar in format. The subroutines are summarized in the following sections.

Analog

AISQW	–	Reads any number of analog input points in the sequence of the hardware interface
AIRDW	–	Reads analog input points in a sequence specified by the user
AOW	–	Writes analog output registers in a sequence specified by the user

The format of the calling sequence is:

```
CALL AISQW (i,j,k,m)
CALL AIRDW (i,j,k,m)
CALL AOW (i,j,k,m)
```

where:

- i specifies the number of analog input/output points to be read or written
- j specifies hardware or software acquisition, conversion, or transmission information for analog input/output points
- k designates an array name which either contains analog input/output values or stores the converted analog input/output values
- m return code indicates either a successful call or an error condition

Digital

DIW	–	Reads digital input registers
DOMW	–	Sets (turns on)user-selected DO points and, after a user-specified length of time, resets (turns off) the selected DO points
DOLW	–	Sets or resets user selected DO groups

The format of the calling sequence is:

```
CALL DIW (i,j,k,m)
CALL DOMW (i,j,k,l,m)
CALL DOLW (i,j,k1,k2,m)
```

where:

- i specifies the number of digital input/output words to be either read, written, or latched
- j specifies hardware or software acquisition, conversion, or transmission information for each digital input/output word
- k designates an array name which either contains the image words to be output or stores the converted digital input/output words
- k_1 designates an array name whose contents are the image words to be output
- k_2 designates an array whose contents define digital outputs which can be changed by the subroutine
- l designates the number of millisecond intervals that are to occur between setting and resetting of the groups
- m return code indicates either a successful call or an error condition

System Service Interface Subroutines

FORTRAN IV provides access to system services through interface subroutines. These subroutines are entered via a CALL and require special parameter types, which are described in the *Series/1 FORTRAN IV: User's Guide*. Supported functions are as follows:

$ATACH	–	Attach a new task
$AWAIT	–	Wait on completion of an event
$CON	–	Connect issuing task to PI interrupt
$DEQUE	–	Remove an element from a queue
$DFNEV	–	Define an event
$DFNQU	–	Define a storage queue
$DFNRS	–	Define a resource
$DISCN	–	Disconnect from PI interrupt
$DLTEV	–	Delete an event
$DLTQU	–	Delete a queue
$DLTRS	–	Delete resource
$DTACH	–	Detach (terminate) task
$ENQUE	–	Add an element to a queue
$MDSST	–	Modify system scheduler table
$MDSTT	–	Modify system task set table
$POST	–	Post completion of an event
$RDTOD	–	Read time-of-day

$RELRS	–	Release resource
$REQRS	–	Request resource
$SERXT	–	Set task error exit
$SETRL	–	Set ROLLIN/ROLLOUT status
$TSQUE	–	Queue task set for execution
$TSSTP	–	Terminate task set execution
$WRTOD	–	Write time-of-day

Appendix C. Non-Standard Integer Lengths with the NOCMPAT Option

If a FORTRAN IV program is compiled with the NOCMPAT option, every integer—whether constant, variable, array, or function—is compiled as INTEGER*2, unless explicitly declared by you as INTEGER*4 in a type statement. (The INTEGER*2 length for constants cannot be overridden.) The NOCMPAT option affects all standard definitions given in the main body of this manual for appearances of integers. Thus, in the READ, WRITE, FIND, END FILE, BACKSPACE, computed GO TO, ASSIGN, assigned GO TO, and RETURN statements, the integers used may be either INTEGER*2 or INTEGER*4.

As explained in Appendix B, the bit manipulator and interrogator functions are only available when the NOCMPAT option is used.

The maximum magnitude of an INTEGER*2 number is 32767.

Appendix D. Debug Facility

The debug facility is a programming aid that enables you to locate errors in a FORTRAN IV source program. The debug facility provides for tracing, at execution time, the flow within a program and between programs.

The debug facility consists of a DEBUG specification statement, an AT debug packet identification statement, and two TRACE statements (TRACE ON and TRACE OFF). These statements are used to define the desired debugging operations for a single program unit in source language. (A program unit is a single main program or a subprogram.)

The DEBUG specification statement is similar to the END statement in that the statement preceding it must follow the same rules as those for the END statement. Thus, a STOP statement, an arithmetic IF, a GO TO, or a RETURN statement is required.

The source deck arrangement consists of the source language statements that constitute the program, followed by the DEBUG specification statement, followed by the debug packets, followed by the END statement.

The statements that make up a program debugging operation must be grouped in one or more debug packets. A debug packet consists of an AT specification statement, followed by a TRACE ON or TRACE OFF statement and/or FORTRAN IV source language statements, and is terminated by either another debug packet or the END statement of the program unit.

DEBUG Facility Statements

The specification statement (DEBUG) sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking). The debug packet identification statement (AT) identifies the beginning of the debug packet and the statement in the program at which tracing is to begin. The two executable statements (TRACE ON and TRACE OFF) designate actions to be taken at specific points in the program.

DEBUG Specification Statement

There must be one and only one DEBUG statement for each program or subprogram to be debugged, and it must immediately precede the first debug packet.

The options in a DEBUG specification statement may be given only once, may appear in any order, and must be separated by commas.

General Form of the DEBUG Statement

```
DEBUG list
```

where:

- list is optional and may contain one or both of the following (with a separating comma, if both):

`TRACE` This option must be in the DEBUG specification statement of each program or subprogram for which tracing is desired. If this option is omitted, there can be no display of program flow by statement number within this program. Even when this option is

	used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.
`SUBTRACE`	This option specifies that the name of the subprogram in which this DEBUG statement appears is to be displayed whenever it is entered. A message is to be displayed whenever execution of the subprogram is completed.

AT Debug Packet Identification Statement

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging is to begin. There must be one AT statement for each debug packet; there may be many debug packets for one program or subprogram.

General Form of the AT Statement

```
AT statement number
```

where:

- statement number is an executable statement number in the program or subprogram to be debugged.

The debugging operations specified within the debug packet are performed immediately prior to the execution of the statement indicated by the statement number in the AT statement.

TRACE ON Statement

The TRACE ON statement initiates the display of program flow by statement number. Each time a statement with an external statement number is executed, a record of the statement number is made on the debug output file. This statement has no effect unless the TRACE option is specified in the DEBUG specification statement.

General Form of the TRACE ON Statement

```
TRACE ON
```

For a given debug packet, the TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement; tracing continues until a statement referred to by a TRACE OFF is encountered. The TRACE ON stays in effect through any level of subprogram call or return. However, if a TRACE ON statement is in effect and control is given to a program in which the TRACE option was not specified, the statement numbers in that program are not traced. Trace output is placed in the debug output file.

This statement may not appear as the conditional part of a logical IF statement.

TRACE OFF Statement

The TRACE OFF statement may appear anywhere within a debug packet and stops the recording of program flow by statement number.

General Form of the TRACE OFF Statement

```
TRACE OFF
```

This statement may not appear as the conditional part of a logical IF statement.

Programming Considerations

The following precautions must be taken when setting up a debug packet:

- Any DO loops initiated within a debug packet must be wholly contained within that packet.
- Statement numbers within a debug packet must be unique. They must be different from statement numbers within other debug packets and within the program being debugged.
- An error in a program should not be corrected with a debug packet; when the debug packet is removed, the error remains in the program.
- The following statements must not appear in a debug packet:
  PROGRAM
  SUBROUTINE
  FUNCTION
  ENTRY
  IMPLICIT
  COMMON
  GLOBAL
  EQUIVALENCE
  BLOCK DATA
  INVOKE
  statement function definition
  type statement
- The program being debugged must not transfer control to any statement number defined in a debug packet; however, control may be returned to any point in the program from a packet. In addition, a debug packet may contain a RETURN or STOP statement.
- If debugging is desired in any subprogram, the main program must also contain a DEBUG statement.

Programming Examples

The following examples show the use of a debug packet to test the operation of a program.

Examples:

```
     INTEGER SOLON, GFAR, EWELL
     •
     •
     •
10   SOLON = GFAR * SQRT(FLOAT(EWELL))
11   IF (SOLON) 40, 50, 60
     •
     •
     •
     DEBUG
     AT 11
     WRITE (1,21) GFAR, SOLON, EWELL
21   FORMAT (1X,'GFAR=',I10,'SOLON=',I10,'EWELL=',I10)
     END
```

The values of SOLON, GFAR, and EWELL are to be examined as they were at the completion of the arithmetic operation in statement 10. Therefore, the statement number entered in the AT statement is 11.

The debugging operation indicated is carried out just before execution of statement 11. If statement number 10 had been entered in the AT statement, the values of SOLON, GFAR, and EWELL would be written as they were before execution of statement 10.

```
    DIMENSION STOCK(1000),OUT(1000)
    •
    •
    •
    DO 30 I = 1, 1000
25  STOCK (I) = STOCK (I) - OUT (I)
30  CONTINUE
35  A = B + C
    •
    •
    •
    DEBUG
    AT 35
    WRITE(1,20) STOCK
20  FORMAT STOCK('VALUES OF STOCK ARE'/(4E16.8))
    END
```

The value of each element in the array STOCK is to be displayed. When statement 35 is encountered, the debugging operation designated in the debug packet is executed. The values of STOCK at the completion of the DO loop are written out.

```
10  A = 1.5
12  L = 1
15  B = A + 1.5
20  DO 22 I = 1.5
    •
    •
    •
22  CONTINUE
25  C = B + 3.16
30  D = C/2
    STOP
    •
    •
    •
    DEBUG TRACE
C   DEBUG PACKET NUMBER 1
    AT 10
    TRACE ON
C   DEBUG PACKET NUMBER 2
    AT 20
    TRACE OFF
C   DEBUG PACKET NUMBER 3
    AT 30
    TRACE ON
    END
```

When statement 10 is encountered, tracing begins as indicated by debug packet 1. When statement 20 is encountered, tracing stops as indicated by the TRACE OFF statement in debug packet 2. When statement 30 is encountered, debug packet 3 commences tracing again.

In this example, trace output is produced for statement numbers 10, 12, 15, and 30. No debug output is produced for statement numbers 20, 22, and 25.

Sample Program 1

This sample program (Figure E-1) is designed to find all of the prime numbers between 2 and 1000. A prime number is an integer greater than 1 that cannot be evenly divided by any integer except itself and 1. Thus 2, 3, 5, 7, 11, ... are prime numbers. The number 9 is not a prime number since it can be evenly divided by 3.

IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM 1

```
C PRIME NUMBER GENERATOR
      MOD(I,J)=I-(I/J)*J
      DATA II /1/
      WRITE (II,1)
    1 FORMAT ('1FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1ØØØ'/
     *19X,'2'/19X,'3'/19X,'5'/19X,'7')
      DO 4 I=11,1ØØØ,2
      K=SQRT(FLOAT(I))
      DO 2 J=3,K,2
      IF (MOD(I,J) .EQ. Ø) GO TO 4
    2 CONTINUE
      WRITE (II,3) I
    3 FORMAT (I2Ø)
    4 CONTINUE
      WRITE (II,5)
    5 FORMAT (' THIS IS THE END OF THE LIST')
      STOP
      END
```

Figure E-1. Sample program 1

Sample Program 2

The n points (x_i, y_i) are to be used to fit an m-degree polynomial by the least-squares method.

$y = a_0 + a_1 x + a_2 x^2 + \bullet \bullet \bullet \bullet + a_m x^m$

In order to obtain the coefficients $a_0, a_1, \ldots, a_m$, it is necessary to solve the normal equations:

(1) $w_0 a_0 + w_1 a_1 + \bullet \bullet \bullet \bullet + w_m a_m = z$

(2) $w_1 a_0 + w_2 a_1 + \bullet \bullet \bullet \bullet + 2_{m+1} a_m = z_1$

•

•

•

(m+1) $w_m a_0 + w_{m+1} a_1 + \bullet \bullet \bullet \bullet + w_{2m} a_m = z_m$

where:

$$w_0 = n \qquad z_0 = \sum_{i=1}^{n} y_i$$

$$w_1 = \sum_{i=1}^{n} x_i \qquad z_1 = \sum_{i=1}^{n} y_i x_i$$

$$w_2 = \sum_{i=1}^{n} x_i^2 \qquad z_2 = \sum_{i=1}^{n} y_i x_i^2$$

•

•

•

$$z_m = \sum_{i=1}^{n} y_i x_i^m$$

•

•

•

$$w_{2m} = \sum_{i=1}^{n} x_i^{2m}$$

After the w's and z's have been computed, the normal equations are solved by the method of elimination which is illustrated by the following solution of the normal equations for a second-degree polynomial (m = 2).

(1) $w_0 a_0 + w_1 a_1 + w_2 a_2 = z_0$

(2) $w_1 a_0 + w_2 a_1 + w_3 a_2 = z_1$

(3) $w_2 a_0 + w_3 a_1 + w_4 a_2 = z_2$

The forward solution is as follows:

1. Divide equation (1) by w_0.
2. Multiply the equation resulting from step 1 by w_1 and subtract from equation (2).
3. Multiply the equation resulting from step 1 by w_2 and subtract from equation (3).

The resulting equations are:

(4) $a_0 + b_{12}a_1 + b_{13}a_2 = b_{14}$

(5) $b_{22}a_1 + b_{23}a_2 = b_{24}$

(6) $b_{32}a_1 + b_{33}a_2 = b_{34}$

where:

$b_{12} = w_1/w_0$, $b_{13} = w_2/w_0$, $b_{14} = z_0/w_0$

$b_{22} = w_2-b_{12}w_1$, $b_{23} = w_3-b_{13}w_1$, $b_{24} = z_1-b_{14}w_1$

$b_{32} = w_3-b_{12}w_2$, $b_{33} = w_4-b_{13}w_2$, $b_{34} = z_2-b_{14}w_2$

Steps 1 and 2 are repeated using equations (5) and (6), with b_{22} and b_{32} instead of w_0 and w_1. The resulting equations are:

(7) $a_1 + c_{23}a_2 = c_{24}$

(8) $c_{33}a_2 = c_{34}$

where:

$c_{23} = b_{23}/b_{22}$, $c_{24} = b_{24}/b_{22}$

$c_{33} = b_{33}-c_{23}b_{32}$, $c_{34} = b_{34}-c_{24}b_{32}$

The backward solution is as follows:

(9) $a_2 = c_{34}/c_{33}$ from equation (8)

(10) $a_1 = c_{24}-c_{23}a_2$ from equation (7)

(11) $a_0 = b_{14}-b_{12}a_1-b_{13}a_2$ from equation (4)

Figure I-5 is a sample FORTRAN program for carrying out the calculations for the case: $n = 100$, $m \leqslant 10$. $w_0, w_1, w_2, \ldots, w_{2m}$, are stored in W(1), W(2), W(3), . . . , W(2M+1), respectively. z_0, $z_1, z_2, \ldots, z_m$ are stored in Z(1), Z(2), Z(3), . . . , Z(M+1), respectively.

IBM FORTRAN Coding Form

SAMPLE PROGRAM 2 1 3

```
C     CURVE-FITTING PROGRAM
      REAL X(1ØØ),Y(1ØØ),W(21),Z(11),A(11),B(11,12)
    1 FORMAT (I2,I3/(4F14.7))
    2 FORMAT (5E15.6)
      READ (5,1) M,N,(X(I),Y(I),I=1,N)
      LW = 2*M+1
      LB = M+2
      LZ = M+1
      DO 5  J=2,LW
    5 N(J) = Ø.Ø
      W(1) = N
      DO 6  J=1,LZ
    6 Z(J) = Ø.Ø
      DO 16  I=1,N
      P = 1.Ø
      Z(1) = Z(1)+Y(I)
      DO 13  J=2,LZ
      P = X(I)*P
      W(J) = W(J)+P
   13 Z(J) = Z(J)+Y(I)*P
      DO 16  J=LB,LW
      P = X(I)*P
```

Figure E-2. (Part 1 of 3) Sample program 2

IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM 2 — PAGE 2 OF 3

```
   16 W(J) = W(J)+P
   17 DO 2Ø  I=1,LZ
      DO 2Ø  K=1,LZ
      J = K+I
   20 B(K,I) = W(J-1)
      DO 22  K=1,LZ
   22 B(K,LB) = Z(K)
   23 DO 31  L=1,LZ
      DIVB = B(L,L)
      DO 26  J=L,LB
   26 B(L,J) = B(L,J)/DIVB
      I1 = L+1
      IF (I1-LB) 28,33,33
   28 DO 31  I=I1,LZ
      FMULTB = B(I,L)
      DO 31  J=L,LB
   31 B(I,J) = B(I,J)-B(L,J)*FMULTB
   33 A(LZ) = B(LZ,LB)
      I = LZ
   35 SIGMA = Ø.Ø
      DO 37  J=I,LZ
```

*A standard card form, IBM electro 888157, is available for punching statements from this form.

Figure E-2. (Part 2 of 3) Sample program 2

IBM FORTRAN Coding Form

PROGRAM SAMPLE PROGRAM 2 — PAGE 3 OF 3

```
   37 SIGMA = SIGMA+B(I-1,J)*A(J)
      I = I-1
      A(I) = B(I,LB)-SIGMA
   4Ø IF (I-1) 41,41,35
   41 WRITE (6,2) (A(I),I=1,LZ)
      STOP
      END
```

Figure E-2. (Part 3 of 3) Sample program 2

The elements of the W array, except W(1), are set equal to zero. W(1) is set equal to N. For each value of I, X(I) and Y(I) are selected. The powers of X(I) are computed and accumulated in the correct W counters. The powers of X(I) are multiplied by Y(I), and the products are accumulated in the correct Z counters. In order to save machine time when the object program is being run, the previously computed power of X(I) is used when computing the next power of X(I). Note the use of variables as index parameters. By the time control has passed to statement 17, the counters have been set as follows:

$$W(1) = N \qquad Z(1) = \sum_{I=1}^{N} Y(I)$$

$$W(2) = \sum_{I=1}^{N} X(I) \qquad Z(2) = \sum_{I=1}^{N} (Y(I)X(I)$$

$$W(3) = \sum_{I=1}^{N} X(I)^2 \qquad Z(3) = \sum_{I=1}^{N} Y(I)X(I)^2$$

$$\vdots \qquad \vdots$$

$$Z(M+1) = \sum_{I=1}^{N} Y(I)X(I)^m$$

$$\vdots$$

$$W(2M+1) = \sum_{I=1}^{N} X(I)^{2m}$$

By the time control has passed to statement 23, the values of $w_0, w_1,\ldots,w_{2m}$ have been placed in the storage locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the values of $z_0, z_1,\ldots,z_m$ have been stored in the locations corresponding to the column M+2 of the B array. For example, for the illustrative problem (M=2), columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

w_0	w_1	w_2	z_0
w_1	w_2	w_3	z_1
w_2	w_3	w_4	z_2

This matrix represents equations (1), (2), and (3), the normal equations for M=2.

The forward solution, which results in equations (4), (7), and (8) in the illustrative problem, is carried out by statements 23 through 31. By the time control has passed to statement 33, the coefficients of the A(I) terms in the M+1 equations which would be obtained in hand calculations have replaced the contents of the locations corresponding to columns 1 through M+1, rows 1 through M+1, of the B array, and the constants on the right-hand side of the equations have replaced the contents of the locations corresponding to column M+2, rows 1 through M+1, of the B array. For the illustrative problem, columns 1 through 4, rows 1 through 3, of the B array would be set to the following computed values:

1	b_{12}	b_{13}	b_{14}
0	1	c_{23}	c_{24}
0	0	c_{33}	c_{34}

This matrix represents equations (4), (7), and (8).

The backward solution, which results in equations (9), (10), and (11) in the illustrative problem, is carried out by statements 33 through 40. By the time control has passed to statement 41, which prints the values of the A(I) terms, the values of the M+1 A(I) terms have been stored in the M+1 locations of the A array. For the illustrative problem, the A array would contain the following computed values for a_2, a_1, and a_0 respectively.

Location	*Contents*
A(3)	c_{34}/c_{33}
A(2)	$c_{24}-c_{23}a_2$
A(1)	$b_{14}-b_{12}a_1-b_{13}a_2$

The resulting values of the A(I) terms are then printed according to the format specification in statement 2.

Appendix F. Comparison with Other FORTRAN IVs

The following Series/1 FORTRAN IV features are not available in IBM Basic FORTRAN IV:

ASSIGN
Assigned GO TO
BLOCK DATA
DATA
Debug facility
ENTRY
ERR and END parameters in a READ
ERR parameter in a WRITE
EXECUTIVE functions
Generalized subscript form
IMPLICIT
Initial data values in explicit specification statements
INVOKE
L-and Z-format codes
Labeled COMMON
Length of variables and arrays as part of type specifications
Literal as actual argument in CALL and function reference
LOGICAL
Logical, literal, and hexadecimal constants
Logical IF
PAUSE with literal
Process I/O subroutines
RETURN i (i not a blank)
Up to seven dimensions in an array

The following Series/1 FORTRAN IV features are not available in American National Standard (ANS) FORTRAN:

Direct-access input/output statements
ENTRY
ERR and END parameters in a READ
ERR parameter in a WRITE
Function names in type statements
Generalized subscripts
GLOBAL
Hexadecimal constant
IMPLICIT
Initial data values in explicit specification statements
Integer*2 data type
INVOKE
Length of variables and arrays as part of type specifications
Length specification in type statement
Literal as actual argument in function reference
Literal enclosed in apostrophes
Mixed-mode expressions
Multiple exponentiation without parentheses to indicate order of computation
PAUSE "message'
PROGRAM
RETURN i
T-and Z-format codes
Up to seven dimensions in an array

The following Series/1 FORTRAN IV features are not available in IBM System/360 and System/370 full FORTRAN IV:

ERR parameter in a WRITE
GLOBAL
PROGRAM
INVOKE

The following IBM System/360 and System/370 full FORTRAN IV features are not available in Series/1 FORTRAN IV:

LOGICAL*1
COMPLEX
NAMELIST
Adjustable (object-time) dimensioning
PRINT
PUNCH
READ b,list
DEBUG with UNIT, INIT, or SUBCHK
DISPLAY
Exponentiation or function reference in subscripts
appearing in I/O lists
Subprogram dummy arguments enclosed in slashes FORMAT with G specification, or more than one level of parentheses, or adjustable (object time) format specification.

In addition, the following FORTRAN IV H Extended features are not supported: asynchronous I/O, extended precision, EXTERNAL extension, and GENERIC. A subset of the FORTRAN IV H Extended list-directed I/O is not supported.

All arguments in Series/1 FORTRAN IV are passed by name. To ensure compatible results when using the same program with System/360 and Series/1 full FORTRAN IV compilers, all arguments in FUNCTION, SUBROUTINE, and ENTRY statements should be enclosed in slashes.

Appendix G. 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 and defined in a DIMENSION, COMMON, GLOBAL, or explicit specification 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 variable or array element, followed by an equal sign (=), followed by an arithmetic expression.

basic real constant. A string of decimal digits containing a decimal point.
blank common. An unlabeled (unnamed) common block.
blank global. An unlabeled (unnamed) global block.

common block. A storage area that may be referred to by a calling program and one or more subprograms.
compilation time. The point in time during which a source program is compiled, that is, translated from a high level language to a machine language program.
compile. To prepare a machine language program from a computer program written in a higher level programming language.
constant. A fixed and unvarying quantity. The four classes of constants specify numbers (numerical constants), logical values (logical constants), literal data (literal constants), and hexadecimal data (hexadecimal constants).
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 IV statements or to terminate the execution of the FORTRAN IV program.

data item. A constant, variable, or array element.
data set. A named collection of data which resides on a device.
data set reference number. A constant or variable used in an input/output statement to identify the data set which is to be operated upon.
data type. The mathematical properties and internal representation of data and functions. The three types are integer, real, and logical.
direct-access file. A file from which records may be retrieved, or to which records may be written, in a nonsequential manner.
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.
double precision. Pertaining to the use of two computer words to represent a number.
dummy argument. A variable within a FUNCTION, SUBROUTINE, or ENTRY statement, or statement function definition, with which actual arguments from the CALL statement 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 IV-defined external procedures or both.
executable statement. A statement which specifies action to be taken by the program; for example, 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 group of DO statements and the transfer back into the range of the 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 IV statements.

file. An ordered collection of one or more records; a data set.
formatted record. A record which is transmitted with the use of a FORMAT statement.
function subprogram. An external function defined by FORTRAN IV statements and headed by a FUNCTION statement. It returns a value to the calling program unit at the point of reference.

global area. A data area that permits communication between two or more programs.

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.

labeled global. A named global 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.

literal constant. A string of alphameric and/or special characters enclosed in quotation marks or preceded by a wH specification.

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 irreducible 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 not containing a FUNCTION, BLOCK DATA, or SUBROUTINE statement and containing at least one executable statement. A main program is required for program execution.

name. A string of from one to six alphameric characters, the first of which must be alphabetic, used to identify a variable, an array, a function, a subroutine, an entry point, or a common or global block.

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 operands, editing information, statement functions, or data arrangement.

numeric character. Any one of the set of characters 0, 1, 2, ..., 9.

numeric constant. An integer or real constant.

object module. A module that is the output of an assembler or compiler and is the input to the application builder.

predefined specification. The FORTRAN IV-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 (see Appendix C); the characters A-H, O-Z, and $ are typed 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 and 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 expresses 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 then 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.

sequential file. A file from which records are retrieved, or to which records are written, solely on the basis of their sequential order.

single precision. Pertaining to the use of one computer word to represent a number.

source program. A computer program written in a source language; for example, a program written in FORTRAN IV.

specification statement. One of the set of statements which provides 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 or global blocks.

statement. The basic unit of a FORTRAN IV program; composed of a line or lines containing some combination of names, operators, constants, or words whose meaning is predefined to the FORTRAN IV compiler. Statements fall into two broad 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 expression.

statement function reference. A reference in an arithmetic or logical expression to a previously defined statement function.

statement number. A number of from one to five decimal digits placed within columns 1 to 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 IV 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 written as an arithmetic constant, arithmetic variable, arithmetic array, or arithmetic expression.

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; transmitted with a one-to-one correspondence between internal storage locations and external positions in the record.

variable. A data item that is not an array or array element; identified by a symbolic name.

Index

FORTRAN IV:
Language Reference
GC34-0133-0

READER'S COMMENT FORM

YOUR COMMENTS, PLEASE . . .

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Cut or Fold Along Line

What is your occupation?____________________

Number of latest Technical Newsletter (if any) concerning this publication: ____________________

Please indicate your name and address in the space below if you wish a reply.

____

____

____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

Your comments, please . . .

This manual is part of a library that serves as a reference source for IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Cut Along Line

Fold

Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

IBM Corporation
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold

Fold

IBM

International Business Machines Corporation
General Systems Division
5775D Glenridge Drive N.E.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

IBM Series/1 FORTRAN IV: Language Reference Printed in U.S.A. GC34-0133-0

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N.E.
P. O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

IBM Series/1 FORTRAN IV: Language Reference Printed in U.S.A. GC34-0133-0

GC34-0133-0