Xerox Data Systems

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox Basic FORTRAN and Basic FORTRAN IV

Sigma 2/3 Computers

Language and Operations

Reference Manual

90 09 67D

August 1970

Price: $2.25

Printed in U.S.A.

REVISION

This publication is a revision of the Xerox Basic FORTRAN/Basic FORTRAN IV Reference Manual for Sigma 2/3 computers, Publication Number 90 09 67C (dated August, 1968). Any changes made to the text from that of the previous manual are indicated by a vertical line in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 2 Computer/Reference Manual	90 09 64
Xerox Sigma 3 Computer/Reference Manual	90 15 92
Xerox Basic Control Monitor (BCM)/BP, RT Reference Manual	90 10 64
Xerox Basic Control Monitor (BCM)/OPS Reference Manual	90 15 06
Xerox Real-Time Batch Monitor (RBM)/RT, BP Reference Manual	90 10 37
Xerox Real-Time Batch Monitor (RBM)/OPS Reference Manual	90 15 55
Xerox Basic FORTRAN/OPS Reference Manual	90 10 61
Xerox Basic FORTRAN IV/OPS Reference Manual	90 15 25
Xerox FORTRAN Library/System Technical Manual	90 10 36

Manual Type Codes: BP - batch processing, LN - language, OPS - operations, RBP - remote batch processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

PREFACE

This manual explains the form and interpretation of programs written in the Basic FORTRAN or Basic FORTRAN IV Programming Languages for use on Xerox Sigma 2/3 Computers. It is a reference guide for programmers and is not intended to be a primer for beginners.

CONTENTS

TABLES

ILLUSTRATIONS

1. INTRODUCTION

FORTRAN is a computer programming system designed to simplify the preparation and checkout of computer programs to solve mathematical or engineering problems. The FORTRAN system consists of two major components:

1. The FORTRAN Language
2. The FORTRAN Processor

Language

The FORTRAN language is a formal language used for specifying computational or information processing procedures for computer execution, concisely and efficiently. It is rigorous and requires the programmer to fully define the characteristics of his problem in a series of statements. These statements normally are written on standard coding forms similar to the one shown in Figure 1. Statement types and formats are discussed in Chapter 2 of this manual, while Chapters 3 and 4 describe syntactic elements of statements and their expressional relationships. Chapters 5 through 9 contain descriptions of the individual statements and the rules for their construction.

Groups of statements form program units. A program unit is either a main program or a subprogram. An executable FORTRAN program consists of precisely one main program and possibly one or more subprograms. Chapter 10 describes the composition of executable programs.

Programs written according to the rules of the FORTRAN language are called source programs.

Processor

The FORTRAN processor is a computer program which, in operation, translates source program statements into computer language instructions. Once compiled, an object program can be loaded and executed on the computer.

The XDS Sigma 2/3 Basic FORTRAN and Basic FORTRAN IV processors compile object programs for the XDS Sigma 2/3 Computers and have the following operating characteristics and capabilities:

1. Compilation is complete after "one pass" of the source program.
2. The processor may be interrupted for priority operations after which FORTRAN processing can continue from the point of interruption.
3. ⊗ The Basic FORTRAN processor operates under control of the Sigma 2/3 Basic Control Monitor, whereas Basic FORTRAN IV operates under control of the Sigma 2/3 Real-Time Batch Monitor.†

In addition to the FORTRAN language and processor, the FORTRAN system provides a library of standard subprograms which may be referenced by FORTRAN programs. These are discussed in Chapter 9, "Subprograms".

A programmer should be familiar with all aspects of the FORTRAN language and the general aspects of the FORTRAN processor in order to take full advantage of the system's capabilities.

---

†Basic FORTRAN IV features are indicated by a ⊗ in the margin of the page.

PROBLEM SAMPLE PROBLEM

PROGRAMMER S. POTTER

XDS

FORTRAN CODING FORM

Identification

73FACTOR 280

PAGE 1 OF 1

DATE

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
		C ROUTINE TO CALCULATE FACTORIALS
		C *****
		K = 1
		READ (1,5) KFACT
10		IF (KFACT) 12, 13, 11
11		K = K * KFACT
		KFACT =
	X	KFACT - 1
		GO TO 10
12		K = 0
13		WRITE (2,6) K
		STOP
5		FORMAT (I6)
6		FORMAT (I20)
		END

XDS-B-44A

Figure 1. Sample Program and Coding Form

2. PROGRAM FORM

A program unit is made up of characters formed into lines and statements.

FORTRAN Character Set

For convenience in explaining various characteristics of the FORTRAN language, the character set used to form statements is divided into four classifications.

Letters

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

No distinction is made between upper and lower case letters. However, for clarity and legibility, exclusive use of upper case letters is recommended.

Digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Strings of digits representing numberic quantities are normally interpreted in the decimal base number system.

Alphanumerics

A subset of characters consisting of all letters and digits.

Special Characters

	Blank	/	Slash
=	Equal	(	Left parenthesis
+	Plus sign	)	Right parenthesis
-	Minus sign	,	Comma
*	Asterisk	.	Decimal point

With the exception of its use in Hollerith fields, a blank character has no meaning and may be used freely to improve legibility of FORTRAN statements.

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

+	Addition or positive value	/	Division
-	Subtraction or negative value	**	Exponentiation
*	Multiplication		

The other special characters have specific applications in the syntactical expression of the FORTRAN language and in the construction of FORTRAN statements.

Any printable character may appear in a Hollerith field.

Lines

FORTRAN lines are strings of 72 characters from the character set.

Line Format

The FORTRAN Coding Form, Figure 1, shows the format of a line. Of the 80 character positions or columns indicated on the form, columns 1 through 72 are used for FORTRAN lines. Columns 73 through 80 are an Identification field, used for sequencing purposes only, and are ignored by the FORTRAN processor.

Columns 1 through 72 comprise three fields:

1. Statement Label (or Number) field — Columns 1 through 5
2. Continuation Character field — Column 6
3. Statement field — Columns 7 through 72

Subsequent discussions will explain the utilization of these fields.

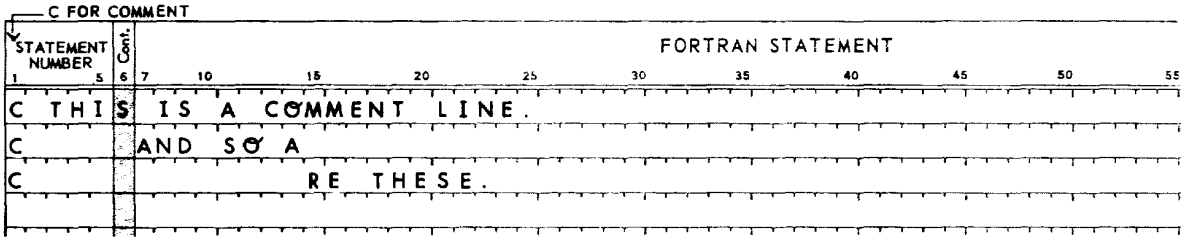
Line Types

Four types of lines are implemented for the FORTRAN language: comment line, END line, initial line, continuation line.

Comment lines are used for source program annotation at the convenience of the programmer. A comment line is identified by the character C in column 1. Columns 2-72 may contain any characters in any desired format.

Comment lines have no effect on the object program and are ignored by the FORTRAN processor except for display purposes in listing the program. A comment line may not be followed by a continuation line.

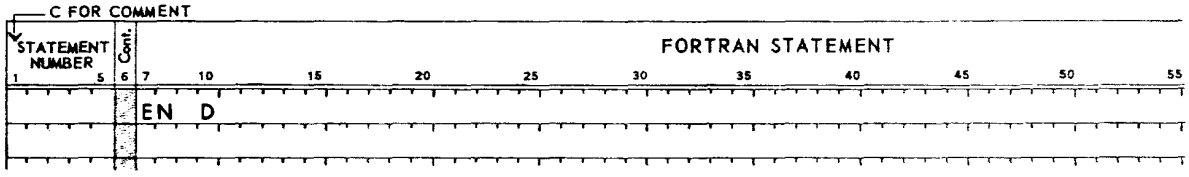
Example:



An END line is the last line of a program unit. Columns 1-6 contain blank characters, and columns 7-72 contain the characters E, N, D, in that order. These characters may be preceded by, interspersed with, or followed by blank characters.

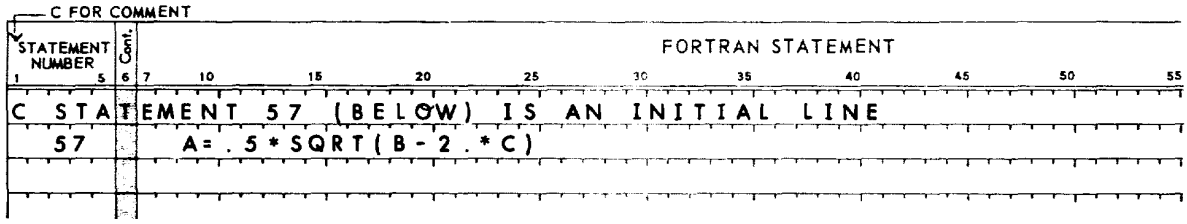
Each FORTRAN program unit must have an END line as its last line to inform the processor that it is at the physical end of the program unit. An END line may follow any other type line.

Example:



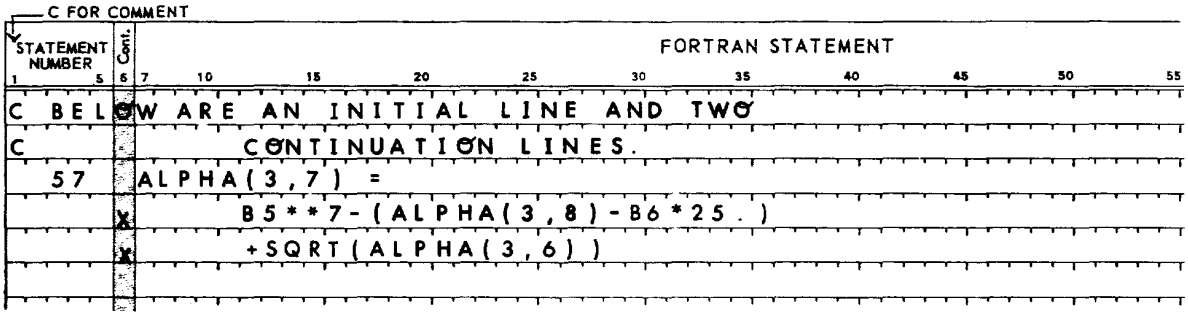
An initial line is the first or only line of each statement. Columns 1-5 may contain a statement label to identify the statement. Column 6 must contain a zero or blank character. Columns 7-72 contain all or part of the statement. An initial line may begin anywhere within the statement field.

Example



Continuation lines are used when additional lines are required to complete a statement originating with an initial line. Columns 1-5 are ignored; column 6 must contain a character other than zero or blank; and columns 7-72 contain the continuation of the statement. There may be as many as six continuation lines in succession. A continuation line may follow only an initial line or another continuation line.

Example:



Statements

A statement consists of an initial line optionally followed by one or more continuation lines.

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or nonexecutable.

Executable Statements

Executable statements specify actions and cause the FORTRAN processor to generate object program instructions. The following classifications of executable statements are implemented:

1. Arithmetic assignment statements
2. Control statements
3. Input/Output statements

Nonexecutable Statements

Nonexecutable statements describe to the processor the nature and arrangement of data, define procedures, and provide information required by the object program during program execution. There are four types of nonexecutable statements:

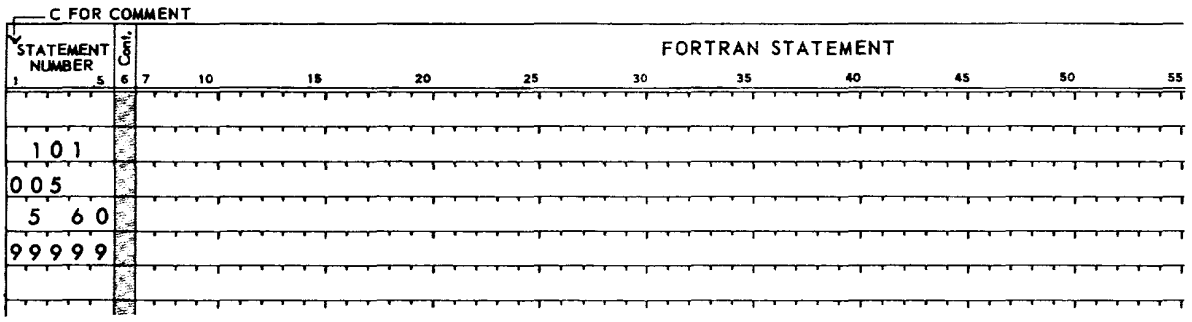
1. Specification statements
2. Format statements
3. Statement Function definitions
4. Subprogram statements

Statement Labels

A statement label may be placed in columns 1-5 of a FORTRAN statement initial line and is used for reference purposes in other statements. The following considerations govern the use of statement labels:

1. The label is an integer from 1 to 99999.
2. The numeric value of the label, leading zeros, or leading, embedded, or trailing blanks are not significant.
3. A label must be unique within a program unit.
4. A label may not appear on a continuation line.

Examples:



Identifiers

Identifiers are used in constructing FORTRAN statements to identify program and information entities. These various entities are discussed in subsequent sections of this manual.

An identifier in the FORTRAN language is a string of from 1 to 6 characters of which the first must be a letter.

Certain sequences of characters which are format field descriptors or which uniquely identify statement types (GO TO, READ, etc.) are not identifiers in such occurrences.

3. DATA TYPES AND IDENTIFICATIONS

The various forms in which data appear in FORTRAN statements are classified by type and name.

Data Types

Two different data types are defined. Each has a different mathematical and language significance and representation. The data types, therefore, have significance in the interpretation of the operations imposed upon them.

Integer data are precise representations of integer values in the range from -32,767 to +32,767.

⊛ Real data are approximations of the values of real numbers — positive, negative, or zero — represented in computer storage in floating point form. Real data in Basic FORTRAN IV may be of two precisions:

1. Standard Precision — precise to 6+ significant digits (i.e., the sixth most significant digit will be accurate, while the seventh will sometimes be accurate, depending on the value) with magnitudes in the range from 5.398×10^{-79} to 7.237×10^{75}.
2. Extended Precision — precise to 9+ significant digits with magnitudes in the range from 10^{-99} to 10^{+99}.

Basic FORTRAN real data is standard precision. Basic FORTRAN IV real data may be either standard or extended precision, but not mixed.

Data Identification

Data, as employed in FORTRAN statements, are identified either by name or value. Rules for representing these data further define four general classifications of data identification: constants, variables, arrays, and array elements.

Constants

Constants are data that do not vary in value and are referenced by naming their values. There are constants for each type of data. Although numeric constants are considered as being unsigned, they may be preceded by the plus or minus operators. The operator is not considered part of the constant, however.

Table 1 gives the rules for the various data types.

Table 1. Constant Formats

Type	Format Rules	Examples
Integer	1. 1 to 5 decimal digits interpreted as a decimal number	-217
	2. No decimal point, comma, or blank characters are allowed	512
	3. Value range: -32,767 to +32,767	0
Real	1. A decimal number represented in one of the forms:	
	a. ±i. ±.f ±i.f	-15. .56321 15.56321
	b. ±i.E±e ±.fE±e ±i.fE±e	73.E02 .32E-04 73.5E03
	where i, f, and e are strings of digits representing integer, fraction, and exponent, respectively.	
	2. A decimal point must be present as shown in 1.	
	3. In the forms shown in 1.b above, if r represents any of the forms preceding E±e (i.e., rE±e) the value is interpreted as r times 10^e.	
	4. If the constant preceding E±e contains more significant digits than the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.	

Variables

Variable data are identified in a FORTRAN statement by identifiers. The names are unique strings of from 1 to 6 alphanumeric characters of which the first must be a letter.

The data type of a variable is specified implicitly by the first character of the variable name as integer or real.

Integer Variable

If the first character of the name is I, J, K, L, M, or N, the variable is typed as integer, unless it is otherwise explicitly typed.

Examples:

L13 K I2 JAZZ MXFEE NEXT

Real Variable

If the first character of the name is other than I, J, K, L, M, or N, the variable is typed as real, unless it is otherwise explicitly typed.

Examples:

AAA BEST FA111 ZAP X

Arrays

An array is an ordered set of data characterized by the property of dimension. Arrays may have one, two, or three dimensions and are denoted by an identifier. Identification of the entire set of data is achieved by the use of the array name. The data typing of an array is accomplished in the same manner as with a variable.

An array name must be declared as such by a DIMENSION statement which also specifies the number of dimensions and size of the array. The DIMENSION statement is discussed in Chapter 6.

Array Elements

An array element is one member of the data set that makes up an array.

Identification of an array element is accomplished by immediately following the array name with subscripts, enclosed in parentheses, which point to a particular element of the array.

(The term array element is synonymous with the term subscripted variable used in some FORTRAN reference manuals.)

Subscripts

Subscripts follow array names to uniquely identify array elements. As used in FORTRAN statements, subscripts assume the same representational meaning as they do in familiar algebraic notation.

Subscripts are constructed and used according to the following rules:

1. No more than three subscripts are allowed.
2. If there are two or three subscripts within the parentheses, they must be separated by commas.
3. The number of subscripts must be the same as the number of dimensions specified in the array declaration.
4. A subscript is written in one of the following forms:

   c*v+k
   c*v-k
   c*v
   v+k
   v-k
   v
   k

   where c and k are integer constants and v is an integer variable name.
5. Subscripts may not be subscripted.

Examples of array elements and subscripts:

X(2*J-3,7) A(I,J) B(20) C(L-2) Y(I)

4. EXPRESSIONS

Expressions are strings of operands separated by operators. Operands may be constants, variables, or function references. Operators may be unary, operating on a single operand, or they may be binary, operating on pairs of operands. All expressions are single valued; the evaluation of any expression has a unique result.

An expression may contain subexpressions. Subexpressions are expressions enclosed in parentheses.

Arithmetic Expressions

An arithmetic expression is a sequence of integer, and/or real, constant, variable, or function references connected by arithmetic operators.

The arithmetic operators are

- `+` Addition or positive value
- `-` Subtraction or negative value
- `*` Multiplication
- `/` Division
- `**` Exponentiation

The appearance of contiguous operators is not allowed. For example, X* - Y is prohibited. X*(-Y) is allowed.

Parentheses may not be used to imply multiplication. The only acceptable indication of multiplication is the asterisk (*) appearing between multiplier and multiplicand. For example, neither CD nor C(D) is acceptable to mean "C times D". It must be expressed C*D or C*(D).

Permissible Expressions

The following rules define all permissible expression forms.

1. A constant, variable name, array element reference, or function reference (Chapter 9) standing alone is an expression.

   Examples:

   S(I) JOBNO 217 17.26 SQRT(A+B)

2. If E is an expression, then +E and -E are called signed expressions, where the + and - are unary operators.

   Examples:

   -S(I) +JOBNO -217 +17.26 -SQRT(A+B)

3. If E is an expression, the form (E) means the evaluated quantity E taken as an entity.

   Examples:

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

4. If E is an unsigned expression and F is any expression, then: F+E, F-E, F*E, F/E, and F**E are all expressions.

   Examples:

   -(B(I, J)+SQRT(A+B(K, L)))

   1.7E-2**(X + 5.0)

   -(B(I+3, 3*J+5)+A)

5. An evaluated expression may be integer or real. The type is determined by the data types of the elements of the expression. All elements must be of the same type with the following exceptions:

   a. A real datum may appear in an integer expression only as an argument of a function.

      Example:

      I + LFUNC(B)

   b. An integer datum may appear in a real expression only as an argument of a function, as a subscript, or as an exponent.

      Examples:

      ABLE + AFUNC(I+2)

      A(I, J+ 1)

      B**N

6. An expression may contain nested parenthesized elements as in:

   A*(Z - ((Y+X)/T))**J

   where X+Y is the innermost element, (Y+X)/T is the next innermost, and Z - ((Y+X)/T) the next.

   In such expressions care should be taken to see that the number of left parentheses and the number of right parentheses are equal.

Expression Evaluation

Arithmetic expressions are evaluated according to the following rules:

1. Parenthesized expression elements are evaluated first. If parenthesized elements are nested, the innermost elements are evaluated, then the next innermost until the entire expression has been evaluated.

2. Within parentheses and/or wherever parentheses do not govern the order or evaluation, the hierarchy of operations in order of precedence is

   a. Function evaluation
   b. Exponentiation
   c. Multiplication and Division
   d. Addition and Subtraction

   Example:

   The expression

   A*(Z-((Y+R)/T))**J+VAL

   is evaluated in the following sequence:

   $Y+R \longrightarrow e_1$
   $(e_1)/T \longrightarrow e_2$
   $Z - e_2 \longrightarrow e_3$
   $e_3{}^{**}J \longrightarrow e_4$
   $A^*e_4 \longrightarrow e_5$
   $e_5 + VAL \longrightarrow e_6$

3. Wherever operations of equal hierarchy are involved, evaluation proceeds from left to right.

   Examples:

Expression	Evaluated as:
W*X/Y*Z	((W*X)/Y)*Z
B**Z-4.*A*C	(B**Z)-((4.*A)*C)
X-Y-Z	(X-Y)-Z
X/Y/Z	(X/Y)/Z
-X**3	-(X**3)

4. The expression X**Y**Z is not allowed. It should be written:

   (X**Y)**Z to mean $(X^{Y})^{Z}$

   or

   X**(Y**Z) to mean $X^{(Y^{Z})}$

5. Use of an array element reference requires the evaluation of its subscript. Subscript expressions are evaluated under the same rules as other expressions.

5. ARITHMETIC ASSIGNMENT STATEMENT

Arithmetic assignment statements assign values to variables or array elements.

These statements are of the form:

v = e

where

v is a variable name or an array element name.

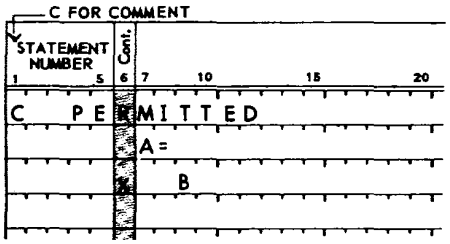
e is an arithmetic expression.

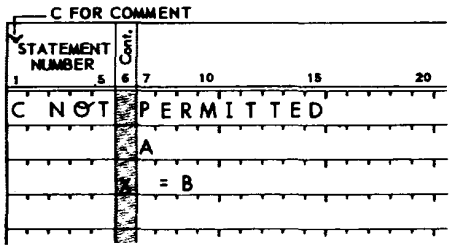
Execution of this statement causes the evaluation of the expression e and the assignment of the resulting value to v.

The following conditions apply to arithmetic assignment statements:

1. Both v and the equality sign must appear on the same statement line.

   Examples:





2. If the data types of v and e are different, the value determined by the evaluation of e will be converted to conform to the data type of v. Table 2 contains the rules for arithmetic assignment.

Table 2. Rules For Arithmetic Assignment

If v Type is	And e Type is	The Assignment Rule is[t]
Integer	Integer	Assign
Integer	Real	Fix and Assign
Real	Integer	Float and Assign
Real	Real	Assign

[t] "Assign" means transmit the resulting value, without change, to the entity.

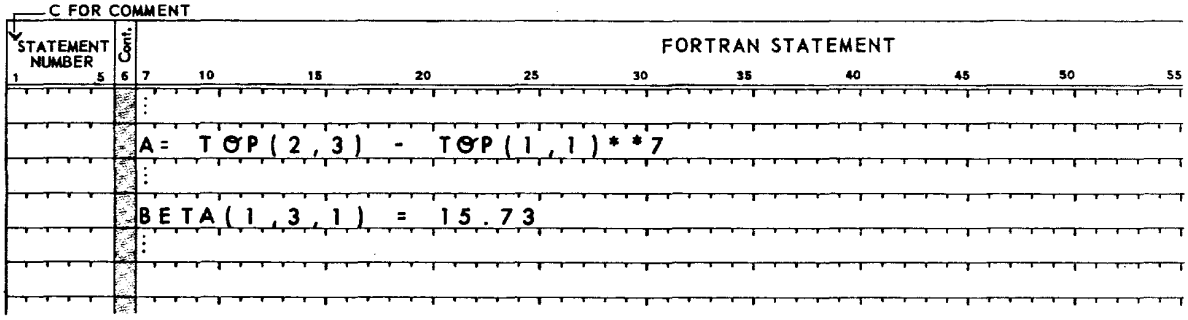
"Fix" means truncate any fractional part of the result and transform that value to the form of an integer datum. For example, in the statement

K = Z

if Z = 56.93, K will have the value 56, and
if Z = -56.93, K will have the value of -56.

"Float" means transform the value to the form of a real datum.

Examples of Arithmetic Assignment Statements:



6. SPECIFICATION STATEMENTS

Specification statements are nonexecutable statements used to inform the FORTRAN processor that certain data referenced in a program are to have specific storage allocation characteristics relative to each other.

There are three types of specification statements:

1. DIMENSION statement – the array declarator statement.
2. COMMON statement – a statement that establishes data storage for common use by two or more program units.
3. EQUIVALENCE statement – a statement that permits the assignment of the same storage by two or more variables.

DIMENSION Statement

A DIMENSION statement is of the form

DIMENSION $v_1(i_1), v_2(i_2), \ldots, v_n(i_n)$

where each v(i) is an array declarator.

Array Declarator

An array declarator specifies the identifier of an array, the number of dimensions (1, 2, or 3), and the size of each dimension.

An array declarator has the form

v(i)

where

- v (the declarator name) is an identifier
- i (the declarator subscript) is composed of an integer constant, or two or three integer constants separated by commas.

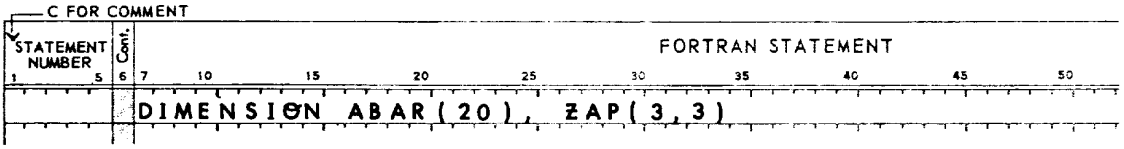
The declarator name followed by declarator subscripts informs the FORTRAN processor that the declarator name is an array name. The number of declarator subscripts specified in (i) indicates the array's dimensions. The values given as subscripts specify the maximum values that subscripts may attain in the array element names.

No array element name may contain a subscript that, during execution of the program in which it appears, assumes a value less than 1 or greater than the maximum dimension length specified in its array declarator.

DIMENSION Statement Rules

1. DIMENSION statements, if they are used, must precede all other statements of a main program unit and may be preceded in a subprogram program unit only by a SUBROUTINE or FUNCTION statement.
2. The number of array declarators in a DIMENSION statement is restricted only by statement length and continuation restrictions.
3. Array declarators may appear only in DIMENSION statements.

   Example:



   This statement declares two arrays:

   a. The 1-dimensional array of 20 elements named ABAR. The array elements are thus named as ABAR(1), ABAR(2), . . . , ABAR(20).

   b. The 2-dimensional array of 9 elements named ZAP. The array elements are ZAP(1, 1), ZAP(2, 1), ZAP(3, 1), ZAP(1, 2), ZAP(2, 2), ZAP(3, 2), ZAP(1, 3), ZAP(2, 3), ZAP(3, 3).

Array Storage Allocation

When a DIMENSION statement is processed by the FORTRAN processor, storage is allocated for the declared arrays. Although actual storage locations for the arrays are not necessarily determined at this point in the processing, the array is henceforth considered as a block in which relative storage is allocated linearly and where the order of ascendency is determined by the first subscript varying most rapidly and the last subscript varying least rapidly. For example, the elements of the array ZAP(3, 3) will be allocated storage in the order in which they appear in statement b above.

Explicit Type Statements

These statements are used to define, explicitly, the type of an identifier. Their form is

REAL $v_1, v_2, \ldots, v_n$

INTEGER $v_1, v_2, \ldots, v_n$

where each v_1 is a variable or array name. When v_1 is an array name, the dimensions may appear with it. These type statements must precede all statements except SUBROUTINE, FUNCTION, and comments.

COMMON Statement

COMMON statements are nonexecutable, storage allocating statements that assign variables and arrays to a storage area called COMMON storage. They provide the facility for various program units to share the use of the same storage area.

COMMON statements are of the form

COMMON $v_1, v_2, \ldots, v_n$

where each v_i is a variable name or array name.

Example:

```
COMMON ZIP, A, B, I, J
```

In this example, the entities ZIP, A, B, I, and J are declared to be in COMMON storage.

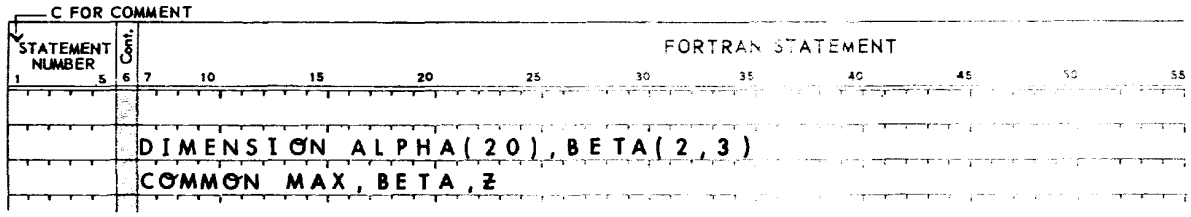
COMMON Statement Rules

1. Each entity listed in a COMMON statement is thus declared to be in COMMON. An entity is either a variable name or array name.
2. More than one COMMON statement may appear in a program unit.
3. The processor strings together in COMMON all entities appearing in the COMMON statements of a program unit in the order of their appearance.
4. If an array name is in a COMMON list, the first element of the array will follow the immediately preceding entity, if one exists, and the last element of the array will precede the next entity, if one exists.
5. The size of COMMON for a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE (see below) statements.
6. The size of COMMON in the various program units that are to be executed together need not be the same.
7. A COMMON statement must precede all statements in a program unit other than DIMENSION, FUNCTION, and SUBROUTINE.
8. Dimension information may be specified in the COMMON statement.
9. A dummy may not appear in a COMMON statement.

COMMON Storage

COMMON storage size is measured in terms of Sigma 2/3 words. In standard precision mode (Basic FORTRAN or Basic FORTRAN IV) each real datum and each integer datum occupy 2 words. In extended precision mode (Basic FORTRAN IV only) each integer datum occupies 1 word and each real datum occupies 3 words.

Example:



These statements define a COMMON area of 8 elements in the order: MAX, BETA(1,1), BETA(2,1), BETA(1,2), BETA(2,2), BETA(1,3), BETA(2,3), Z. In standard precision mode, therefore, the size of COMMON is 14 words; in extended precision mode, the size is 22 words.

Correspondence of COMMON between Program Units

The same data may be referenced by two or more program units through COMMON storage. For all program units COMMON storage allocation starts at the same storage location. In all program units that define the identical type (real or integer) entity to a given position in COMMON (counted by the number of preceding storage words), references to that position refer to the same quantity.

Example:

If a main program contains

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

as its first COMMON statement, and a subprogram contains

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

as its first common statement, then the variables A and W refer to the same data unit. B and X, C and Y, and D and Z have a similar correspondence.

As noted above, for identity between references, the data types of the common data must be the same.

EQUIVALENCE Statements

The use of EQUIVALENCE statements allows two or more entities to share storage.

An EQUIVALENCE statement is of the form

EQUIVALENCE $(k_1),(k_2),\ldots,(k_n)$

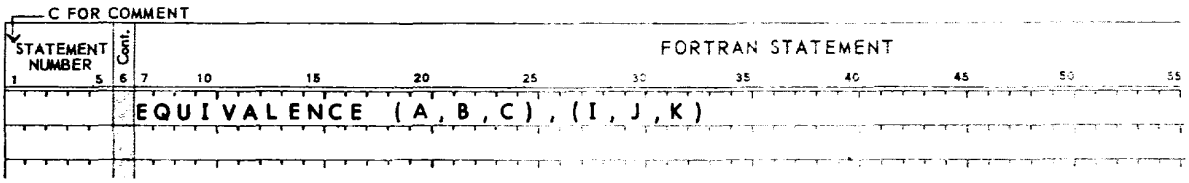
where each k is a list of the form

$u_1, u_2, \ldots, u_m$

Each u is a variable name or an array element name, the subscript of which contains only constants, and m is greater than or equal to 2.

Each element in a list is assigned the same storage position by the processor. The order in which the entities appear is not significant.

Example:



In this example A, B, and C will share the same storage area during object program execution. I, J, and K will also share an identical storage area.

If an array element name is used in an EQUIVALENCE statement, the number of subscripts must be the same as the number of dimensions established by the array declarator, or it must be one number where the one subscript specifies the array element number relative to the first element of the array.

Example:

If the dimension of an array, Z, has been declared as Z(3, 3), then in an EQUIVALENCE statement

Z(6)

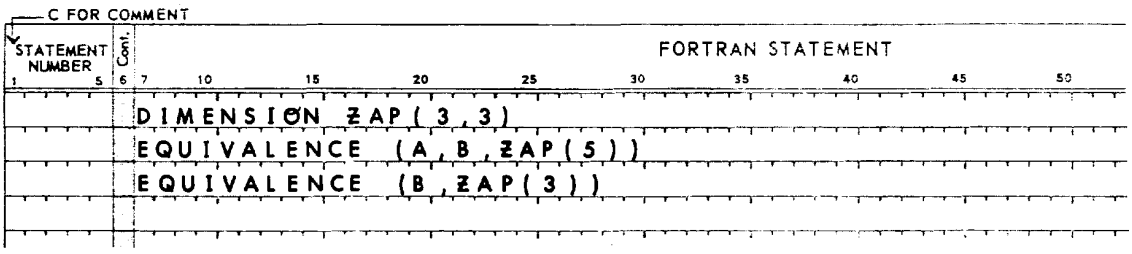
and

Z(3, 2)

have the same meaning.

EQUIVALENCE Statement Rules

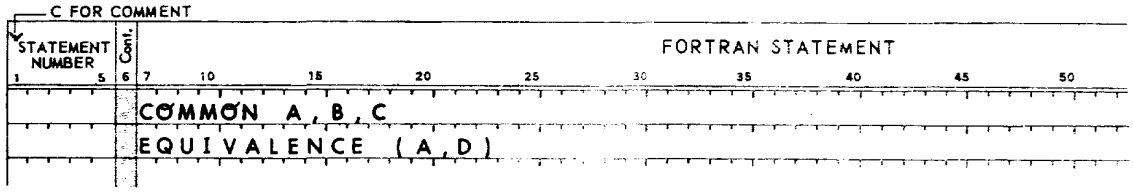
1. EQUIVALENCE statements must not precede any DIMENSION or COMMON statements in a program unit.
2. EQUIVALENCE statements must precede any statements other than DIMENSION, COMMON, FUNCTION, and SUBROUTINE.
3. The subscripts of array element names in EQUIVALENCE statements must be integer constants.
4. An element of a 2- or 3-dimensional array may be referred to by a single subscript, if desired. (See example above.)
5. It is in error to cause, either directly or indirectly, a single storage unit to contain more than one element of the same array. For example, the third statement below is in error for equivalencing B and ZAP(3) when B and ZAP(5) have already been made equivalent.



EQUIVALENCE and COMMON

1. Entities may be assigned to common storage by equating them to previously declared COMMON entities.

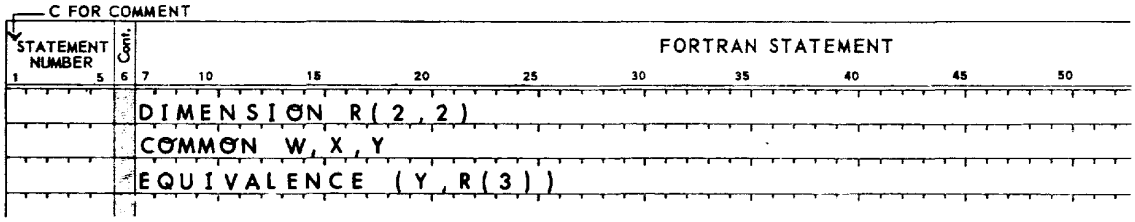
   Example:



   In this case the variables A and D share the first storage position of COMMON storage.

2. EQUIVALENCE statements can increase the size of COMMON storage by adding more entities to the end of the COMMON area.

   Example:



The resulting COMMON storage assignment will have the following arrangement.

Entities

W	R(1,1)	Established by COMMON statement
X	R(2,1)	
Y	R(1,2)	
	R(2,2)	Expanded by EQUIVALENCE statement

Note that

EQUIVALENCE (X,R(3))

would be erroneous in the example. The COMMON statement establishes W as the first entity in COMMON. To make X and R(3) equivalent would then make R(1,1) the first entity and W the second. This is incompatible with the COMMON statement.

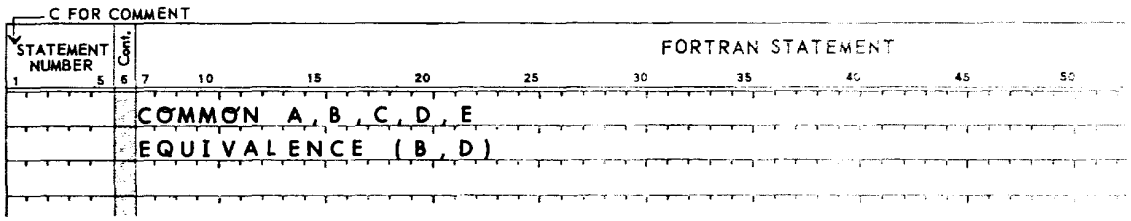
COMMON storage may be increased only forward from the last storage unit established by the COMMON statement, not backward from its first storage unit.

3. When two variables or array elements share storage because of the effects of an EQUIVALENCE statement, the identifiers of the variables or arrays in question may not appear in COMMON statements in the same program unit.

4. It is invalid to equivalence two entities previously assigned to COMMON.

   Example:

   The following statements create an invalid situation:



⊗ EXTERNAL Statement (Basic FORTRAN IV only)

The EXTERNAL statement has the form

EXTERNAL $p_1, p_2, p_3, \ldots, p_n$

where the p_i are subprogram identifiers.

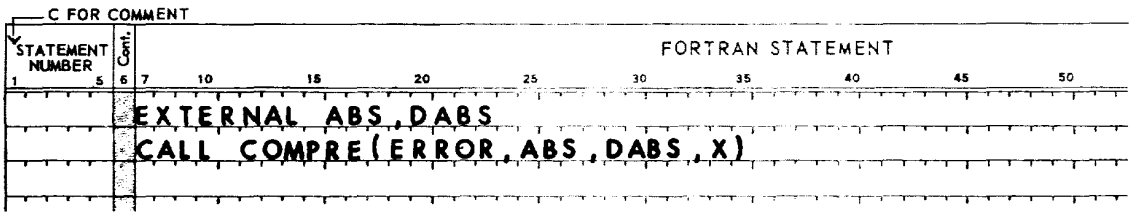
The EXTERNAL statement declares, as subprogram identifiers, names that might otherwise be classified implicitly as scalars, so that they may be passed as arguments to other subprograms. For example, if the subprogram name F appears in the statement

CALL ALPHA(F)

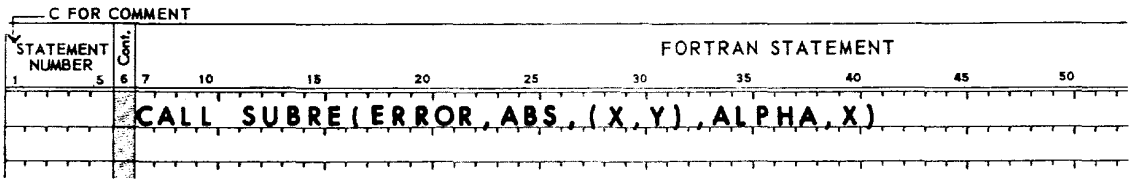
but appears in no other context to indicate that it is a subprogram, it would be implicitly classified as a scalar. The EXTERNAL statement can be used to avoid this.

Examples:

In the following example the subprogram identifiers ABS and DABS are used as arguments in the subprogram COMPRE:



In the following example the subprogram named ABS is not an argument; it is executed first, and its result becomes the argument. In this situation an EXTERNAL statement is not required.



⊙ DATA Statement (Basic FORTRAN IV only)

The DATA statement has the form

DATA $S_1, S_2, S_3, \ldots, S_n$

where

S_i is a data set specification of the form

variable-list/constant-list/

The primary purpose of the DATA statement is to give names to constants. For example, instead of referring to π as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, if a more accurate value is required.

Giving PI a value with a DATA statement is somewhat different from giving it a value with an assignment statement. With the DATA statement the value is assigned when the program is loaded; with the assignment statement, PI receives its value at execution time.

Consider another example that profits even more from the use of the DATA statement: An ARCTAN function can be written using a power series expansion. The efficient way to program this in FORTRAN is with a DO loop, stepping through the constants. But constants cannot be subscripted, and the timing of the routine is adversely affected if an array must be initialized each time into the routine using assignment statements, such as:

C(0) = 0

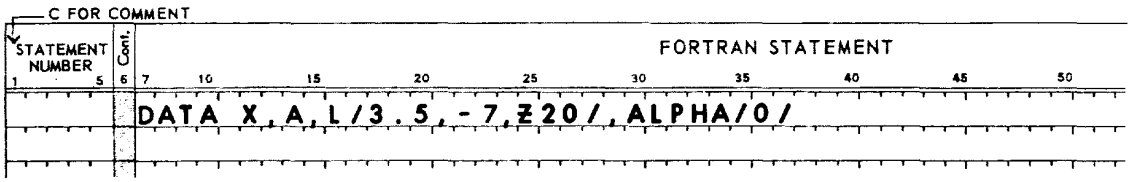
C(1) = .12345 49945

C(2) = .24477 86631

etc.

Here, the DATA statement can be used to great advantage. It is not recommended that the DATA statement be used to give "initial" values to variables that are going to be changed. This causes proper initialization of the program to depend on loading and disallows restarting the program once it has changed these values. Good programming practice dictates that such initialization be done with executable statements, e.g., with assignment statements.

The effect of the DATA statement is to initialize the variables in each data set to the values of the constants in the set, in the order listed. For example, the statement shown below:



is equivalent to the assignment statements

X = 3.5

A = -7

L = 32 (Z20 is a hexadecimal constant; $20_{(16)} = 32_{(10)}$. See following section.)

ALPHA = 0

except that the DATA statement is not executable; its assignments take place upon loading.

Variable and constant lists in DATA statements may be constructed as described in the following two sections.

⊙ DATA Variable List

A DATA variable list is similar to an input list, in that it may contain scalars or subscripted or unsubscripted arrays. It may not contain implied DO loops. Subscripts must be integers.

DATA Constant List

A DATA constant list is of the form

$C_1, C_2, C_3, \ldots, C_m$

where

the C_i are either constants or repeated groups of constants in the following forms:

c

r*c

where

c is a constant of one of the following forms:

1. unsigned integer constant
2. negative integer constant

(See Chapter 3 – may only initialize integer variable.)

3. unsigned real constant
4. negative real constant

(See Chapter 3 – may only initialize real variable.)

5. hexadecimal constant of the form Zx, where x is a string of 1-4 hexadecimal digits. Hexadecimal constants may only initialize integers or variables. The values are right-justified. Examples are: Z12 and ZA7.

6. literal constant of the form:

   'cs'

   where

   cs is a character string. Blanks can be used, and a quote mark can be represented by two consecutive quote marks. Literal constants may initialize either integer or real variables. The length of a literal constant may not exceed 2 characters for integer variables, 4 characters for standard precision real variables or 6 characters for extended precision real variables. Trailing blanks are used to fill out the field if necessary. Examples are

   'ABCD'

   ''''''

   'I J'

r is an unsigned integer repeat count, whose value (nonzero) indicates the number of times the group is to be repeated.

The constant list must completely satisfy the variable list, and there may not be any remaining unused constants. A subscripted array element in the variable list designates the point in the array at which initialization is to begin, and all succeeding elements in the array are to be initialized until the constant list is exhausted.

Example:

```
DIMENSION I(10),J(10)
DATA I(5),J/1, 2, 5*3, 4/
```

initializes the I and J arrays as follows:

I(5) = 1	I(9) = 3
I(6) = 2	I(10) = 3
I(7) = 3	J(1) = 3
I(8) = 3	J(2) = 4

Dummy variables and variables in blank COMMON cannot be initialized with the DATA statement.

7. CONTROL STATEMENTS

Control statements are executable statements used to control and guide the logical flow of FORTRAN programs. The statements in this category are

1. Unconditional GO TO and Computed GO TO statements
2. Arithmetic IF statement
3. DO statement
4. CONTINUE statement
5. PAUSE and STOP statements
6. CALL statement
7. RETURN statement

Statement label references are part of several of these statements. Such statement labels must be associated with executable statements in the same program unit containing the control statement.

GO TO Statements

There are two types of GO TO statements:

1. Unconditional GO TO
2. Computed GO TO

Unconditional GO TO Statement

Unconditional GO TO statements are used in a program whenever control is to be transferred unconditionally to some other statement in the same program unit.

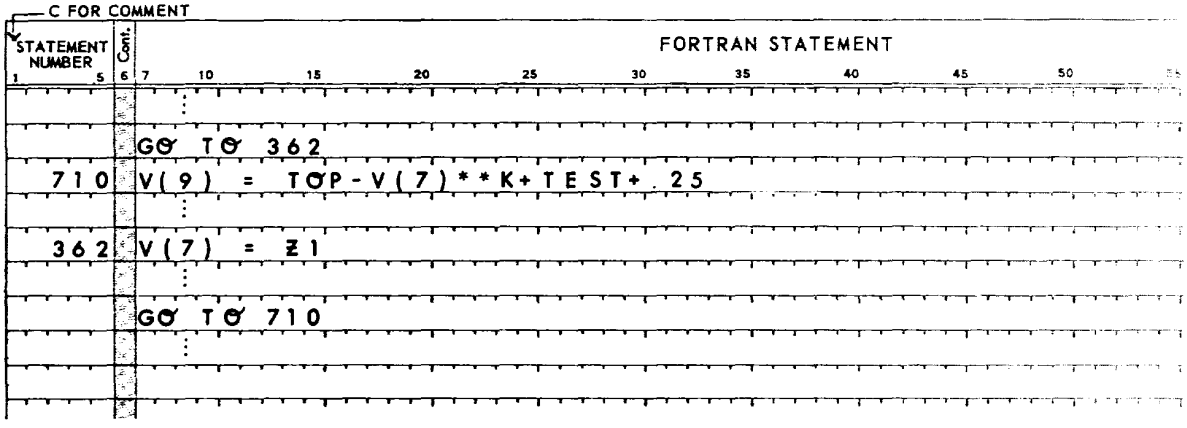
Form:

GO TO k

where

k is the statement label of an executable statement in the same program unit

Example:



In these statements, statement 362 precedes statement 710 in the logical flow of the program of which they are a part, because of the function of the GO TO statements.

Computed GO TO Statement

A computed GO TO statement is a conditional transfer statement which may transfer to any one of several statements depending on a computed condition.

Form:

$$\text{GO TO } (k_1, k_2, \ldots, k_n), \; j$$

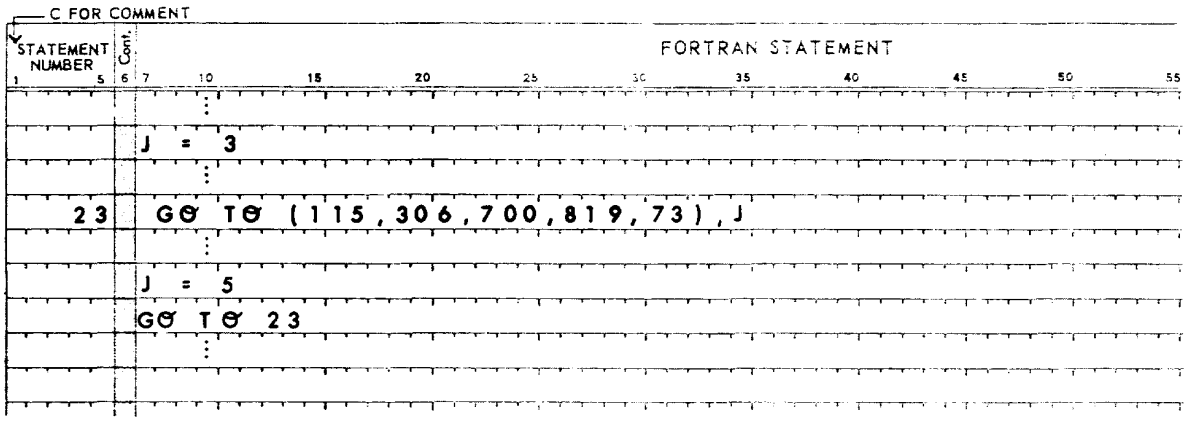
where

k_i are statement labels

j is an integer variable ($1 \leq j \leq n$) that may be altered during program execution to determine a different transfer point for the GO TO statement

This statement causes transfer of control to the statement having a statement label equal to the current value of k_j.

Example:



When J = 3, statement 23 transfers control to statement 700. Changing J to 5 changes any subsequent execution of statement 23 to a transfer to statement 73. Making J = 0 or 6 would be an error because these values are outside the range $1 \leq j \leq n$.

Arithmetic IF Statement

An arithmetic IF statement transfers control in a program to one of three specified statements depending on an expression's value as negative, zero, or positive.

Form:

$$\text{IF(e) } m_1, m_2, m_3$$

where

e is an arithmetic expression (real or integer)

m_1, m_2, m_3 are statement labels

The arithmetic IF statement is interpreted to mean

IF $e < 0$, GO TO m_1
IF $e = 0$, GO TO m_2
IF $e > 0$, GO TO m_3

Examples:

Statement	Expression Value	Transfer to
IF (ZAP) 3,4,5	376.	5
IF (Z-1.) 50,73,9	0.	73
IF (AMTX(3,2)) 7,2,1	-576.	7

DO Statement

The DO statement provides a method for controlled, repetitive execution of a series of statements.

The statement is written in one of the forms:

DO k i = m_1, m_2, m_3

DO k i = m_1, m_2

where

k is the identifier of an executable statement, called the terminal statement of the associated DO

i is an integer variable, called the control variable

m_1 (the initial parameter), m_2 (the terminal parameter), and m_3 (the incrementation parameter) are each either an integer constant or an integer variable reference

If m_3 is not explicitly stated (second form above), a value of 1 is assumed for the incrementation parameter. When a DO statement is executed, m_1, m_2, and m_3 must be greater than zero.

The terminal statement (i.e., statement labeled k) must physically follow and be in the same program unit as the DO statement.

A terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement.

The portion of a DO statement through the first comma (DO k i = m_1,) must always appear in an initial line. The parameters m_2, m_3 may be placed on a continuation line.

DO Range

Associated with each DO statement is a range which is controlled by that DO. The range is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO.

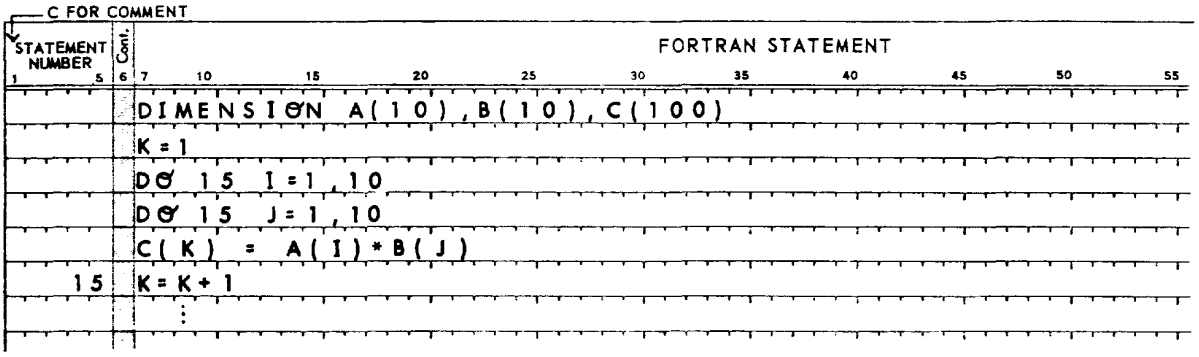
Within the range of a DO statement there may be other DO statements, in which case, the DO ranges must be nested. That is, if the range of one DO contains another DO, then the range of the inner DO must be entirely contained in (be a subset of) the outer DO.

The terminal statement of the inner DO may also be the terminal statement of the outer DO.

Example:

Given the 10-element arrays A and B, compute a 100-element array C such that:

C(1) = A(1)*B(1), C(2) = A(1)*B(2), . . . , C(10) = A(1)*B(10), C(11) = A(2)*B(1), . . . , C(100) = A(10)*B(10).



Execution of the DO Loop

A DO statement defines a loop. A DO loop is executed in the following manner:

1. The control variable, i, is assigned the value represented by the initial parameter, m_1.
2. The range of the DO is executed.
3. If control reaches the terminal statement and the terminal statement is executed, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by its associated incrementation parameter, m_3. $i+m_3$ becomes the new control variable.
4. If this new control variable is less than or equal to m_2, the process is repeated from step 2, using the incremented control variable. If the value of the incremented control variable is greater than the value of its associated terminal parameter, m_2, the DO has been satisfied and the control variable then becomes undefined.
5. If there were other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value of its associated incrementation parameter and the process is repeated from step 4 until all DO statements referring to the terminal statement are satisfied. After this, the first executable statement following the terminal statement is executed.

Other Features of a DO Loop

1. If exit is made from a DO loop other than by satisfying the DO (i.e., with a GO TO or arithmetic IF), the control variable of the DO remains defined and is equal to the latest value attained as defined in "Execution of the DO Loop".
2. No GO TO statement or arithmetic IF statement may cause control to be transferred into the range of a DO statement from outside its range.
3. If a subprogram is referenced in a DO loop, the statements of that subprogram are considered to be temporarily within the range of the DO.
4. The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the DO range.
5. If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be referenced by any GO TO or arithmetic IF statement that occurs anywhere except in the range of the innermost DO having that terminal statement.

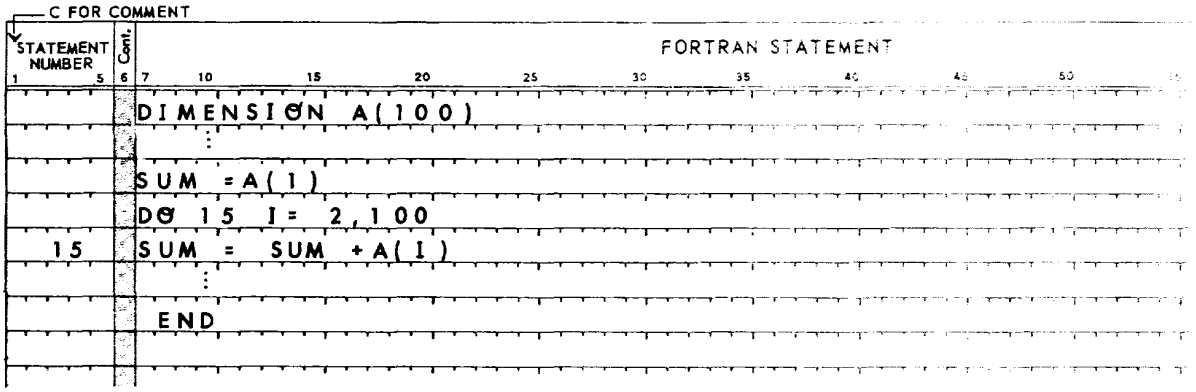
Examples of DO Loops

1. The following example computes

$$\sum_{i=1}^{100} A_i$$

where

A is a 1-dimensional, 100-element array



2. This example computes the 15 elements of the array GAMMA according to the formula

$$GAMMA_m = \sum_{n=1}^{15} ALPHA_{mn} \, BETA_n, \; m = 1, 2, 3, \ldots, 15$$

where

ALPHA is a 2-dimensional array of 15 rows and 15 columns

BETA is a 1-dimensional array of 15 elements

```
C FOR COMMENT
STATEMENT NUMBER   Cont.   FORTRAN STATEMENT
1        5         6       7  10  15  20  25  30  35  40  45  50  55
                   DIMENSION ALPHA(15,15),BETA(15),
                  X        GAMMA(15)
                     :
                   DO 533 M = 1,15
                   GAMMA(M) = 0.0
                   DO 533 N = 1,15
              533  GAMMA(M) = GAMMA(M)+ALPHA(M,N)
                  X        *BETA(N)
                     :
```

CONTINUE Statement

Form:

CONTINUE

Continue is classed as an executable statement. However, it only causes control to go to the next executable statement in logical sequence. It is frequently used as the terminal statement in a DO range when the statement that would normally be the terminal statement is one of those not allowed.

Example:

```
C FOR COMMENT
STATEMENT NUMBER   Cont.   FORTRAN STATEMENT
1        5         6       7  10  15  20  25  30  35  40  45  50  55
                   DIMENSION A(500)
                     :
                   XLRG = A(1)
                   DO 17 I = 2,500
                   IF (XLRG - A(I)) 15,17,17
               15  XLRG = A(I)
               17  CONTINUE
                   GO TO 33
                     :
```

Program Control Statements

FORTRAN provides two program control statements: PAUSE and STOP.

PAUSE Statement

The PAUSE statement may take either of two forms:

PAUSE n

PAUSE

where

n is a decimal digit string from 1 to 4 digits long

A PAUSE statement causes a temporary cessation of program execution and displays "PAUSE n" on the console typewriter. The statement permits the operator to intervene in the program execution for setup and control functions, such as changing data tapes. The operator can then signal the program to continue execution, beginning with the statement immediately after PAUSE.

If no changes are made to the status of the program or the computer during a PAUSE, resuming program execution causes the normal execution sequence to be continued.

Example:

```
PAUSE 123
```

STOP Statement

The STOP statements are written in the forms

STOP n

STOP

where

n is a decimal digit string from 1 to 4 digits long

A STOP statement causes program execution to be terminated (logically the last statement of a program) and displays "STOP n" on the console typewriter.

Example:

```
STOP 1371
```

CALL Statement

A CALL statement is one of the forms:

CALL $s(a_1, a_2, \ldots, a_n)$

CALL s

where

s is the name of a subroutine

a_i are arguments

See "SUBROUTINE Subprograms" in Chapter 9 for a discussion of subroutines, the use of CALL statements, and the characteristics of arguments.

At the beginning of execution of a CALL statement, the designated subroutine, s, is referenced. Such a reference supplies the arguments, a_i, required for executing the subroutine.

When control is returned from the designated subroutine, CALL statement execution is complete.

Examples:

```
CALL DUMP
CALL SUB1(A,B,(I-J),C(100))
CALL SUB2(ARRY,ZAP)
CALL ROUT(X,Z,5,I)
```

RETURN Statement

A RETURN statement is of the form

```
RETURN
```

This statement marks the logical end of a subprogram and may only appear in a subprogram. See Chapter 9 for a discussion of subprograms.

Executing the RETURN statement causes control to return to the current referencing program unit.

If the RETURN statement is executed from within a FUNCTION subprogram, the value of the function (see Chapter 9) is made available to the referencing program.

8. INPUT/OUTPUT

The FORTRAN language provides a series of statements that define the control of and conditions for data transmission between computer storage and external data handling devices such as magnetic tape and paper tape handlers, typewriters, card punch units, line printers, and RADs. These statements are of five types.

1. Sequential READ and WRITE statements that cause specified lists of data to be transmitted between computer storage and any one of a group of external devices.
2. DEFINE FILE statements for defining the characteristics of a direct access file.
3. Direct access READ and WRITE statements that cause specified lists of data to be transmitted between computer storage and a RAD.
4. Auxiliary I/O statements for positioning and demarcation of external files (as on magnetic tapes).
5. FORMAT statements used in conjunction with the input/output of formatted records to provide conversion and editing information that specifies their internal and external representation.

The data transmitted by input/output statements are transmitted as records consisting of binary-coded strings of characters or unformatted binary values in a form similar to internal storage. For either type of transmission the input/output statements refer to external devices, lists of data names, and, for formatted data, to format specification statements.

Sequential Input/Output Statements

The input/output statements described elsewhere in this section all specify a device unit number, u. This number may be either an integer constant or an integer variable reference whose value then identifies the unit. This unit number corresponds to an actual physical device in one of two ways:

1. The number may be assigned to a device at program run-time through the I/O Control System or the Monitor (see XDS Sigma 2/3 FORTRAN Operations Manuals and Monitor Reference Manuals).
2. The number may be a standard unit number assignment, which is recognized as referring to a particular device. (These standard assignments, as well as all others, may be overridden by run-time assignments, if necessary.)

Error checking for valid unit number assignments is performed for all but the auxiliary I/O statements explained later (REWIND, BACKSPACE, ENDFILE). Consequently, if one of the auxiliary I/O statements specifies an invalid unit number (perhaps because of an overriding run-time assignment) no error notification will be given, and the program may execute incorrectly. Table 3 shows the recommended standard device assignments for Sigma 2/3 Basic FORTRAN and Basic FORTRAN IV. Device assignments are made at system generation time.

Table 3. Recommended Standard Unit Assignments

Unit Number	Standard Assignments
101	Typewriter input
102	Typewriter output
103	Paper tape reader
104	Paper tape punch
105	Card reader
106	Card punch
108	Line printer

Formatted Input/Output Statements

Formatted input/output statements are used to process binary-coded (BCD) records. These statements are in the following generalized forms:

READ(u,f)k or READ(u,f)
WRITE(u,f)k or WRITE(u,f)

where

u is a device unit number (unsigned integer or integer variable)

f is a FORMAT reference. It may be the statement label of a FORMAT statement or the name of an array into which a FORMAT statement has been read.

k is an input/output list (or it may be absent)

A formatted READ statement causes a character string in the external record of unit u to be converted into binary values according to the specified FORMAT statement, f, and these values are then assigned to the variables specified in the list, k.

Conversely, a WRITE statement causes internal binary values identified by the variables specified in the list, k, to be converted according to FORMAT statement, f, and output on unit u.

Examples:

```
READ(105,23)RA,RB,RC,RD
WRITE(108,23)ID,VAL
READ(K,15)(ABLE(I),I=1,25)
WRITE(1,17)((ARRAY(I,J),J=1,15),I=1,15)
```

} See DO-implied terms under "List Items".

Formatted Record Processing

Each formatted input/output statement begins processing with a new record. Thus, processing of any record by more than one READ or WRITE statement is restricted. If only part of a record is read, the remainder of the record is skipped. If output is to a device that specifies a fixed-length record and the WRITE statement does not fill the record, the remainder of the record is filled with blanks; otherwise, FORMAT and list specifications determine record length. (See "Formatted Record Sizes" below.)

More than one record may be processed by these statements if specifically requested by the FORMAT statement. However, attempting to read or write more characters on a record than are (or can be) physically contained on the record does not cause a new record to be started. On output the extra characters are lost; on input they are treated as blanks.

Thus, it is necessary for the FORTRAN programmer to recognize the relationships between records and FORMAT specifications.

If the list, k, is omitted from a formatted input/output statement, the normal result is that one record is skipped on input or one blank record is written (on output). An exception to this is when the associated FORMAT statement begins with a Hollerith or slash specification. (See "FORMAT Statements".)

Formatted Record Sizes

A formatted (BCD) record may have a maximum size of 132 characters. Certain devices may impose other restrictions on the size of records (e.g., a punched card contains 80 characters).

A record may contain as few as zero characters, in which case, it is considered to be a blank (or empty) record (e.g., a blank card).

On devices such as magnetic or paper tape, the FORMAT statement may determine the actual size of an output record. (The XDS Sigma 2/3 FORTRAN Operations Manuals and Monitor Reference Manuals contain complete descriptions of BCD records.)

Unformatted (binary) Input/Output Statements

Unformatted (binary) input/output statements transmit information in internal (binary) form and are designed to provide temporary storage on magnetic tapes and discs.

The forms of these statements are

READ (u)k or READ (u)

WRITE (u)k or WRITE (u)

where

u is a device unit number (unsigned integer or integer variable)

k is an input/output list (or it may be absent)

The statements process data as a string of binary digits arranged into words according to the items in the list, k. All data processed by an unformatted READ or WRITE statement are contained in one logical record.

Examples:

```
WRITE (12) ARRAY, BRRAY

READ (11) REC1, CRRAY
```

END and ERR Return Options on READ

These options allow the programmer to specify an exit to a special handling routine when end-of-file or error conditions occur during execution of a sequential (formatted or unformatted) READ statement.

The general forms of these options are

```
READ (u,f, END = b, ERR = b) k

READ (u, END = b, ERR = b) k
```

where

u, f, and k are as defined previously.

b is the number of the statement to which control is transferred in the event that the specified condition occurs.

END specifies that reading an end-of-file mark will cause a special exit.

ERR specifies that the special exit is to be taken upon encountering certain I/O errors (see XDS Sigma 2/3 Basic FORTRAN IV Operations Manual for errors that use the special exit).

Logical Record Form

A logical record may consist of one or more physical records; however, as far as the programmer is concerned, it is treated as a single record. (The XDS Sigma 2/3 FORTRAN Operations Manuals and Monitor Reference Manuals contain complete descriptions of unformatted records.) The records produced by an unformatted WRITE statement contain control words, in addition to the transmitted data, to facilitate reading or backspacing the proper number of physical records.

Unformatted Record Processing

The information output by a single unformatted WRITE statement must be input later by only one READ statement. It is permissible to read less than a full record.

If the input list requires more data than is contained in the record, an error will occur. The number of items that may be processed by an unformatted READ/WRITE statement is not limited, but only one logical record will be processed regardless of the amount of data transmitted.

If the list k is omitted from an unformatted READ/WRITE statement, a record is skipped or a blank record is written. No data can be transferred in such a transmission. Writing such blank records has little purpose, for the record must then be read by a READ statement without a list.

Input/Output List Specifications

An input/output list represents an ordered group of data names that identify the data to be transmitted and the order of their transmission. These lists have the form

$$m_1, m_2, \ldots, m_n$$

where m_i are list items separated by commas, as shown.

List Items

A list item may be a single datum identifier or a multiple data identifier.

A single datum identifier is the name of a variable or array element. One or more of these items may be enclosed in parentheses without changing their intended meaning.

Examples:

READ(5,7)A

READ (6,23) C(26,1), R, K, D, (I,J)

WRITE (1,73) B,I(10,10), S, (R, K), F(1,25)

Multiple data identifiers are in one of two forms:

1. An array name appearing in a list without subscript(s) is considered equivalent to the listing of each element in the array.

   Example:

   If B is a 2-dimensional array, the list item B is equivalent to

   B(1,1), B(2,1), B(3,1), . . . , B(1,2), B(2,2), . . . , B(j,k)

   where j and k are subscript limits of B.

2. DO-implied items are lists of one or more identifiers or other DO-implied items followed by a comma character and an expression of one of the forms:

   $i = m_1, m_2, m_3$

   $i = m_1, m_2$

   and enclosed in parentheses.

   The elements i, m_1, m_2, and m_3 have the same meaning as defined for the DO statement. The items enclosed in parentheses with a DO implication are considered to be in the range of the DO implication. For input lists the indexing parameters, i, m_1, m_2, and m_3, may appear in this range only as subscripts.

Examples:

DO-implied Lists	Equivalent Lists
(X(I), I = 1, 4)	X(1), X(2), X(3), X(4)
(Q(J), R(J), J = 1, 2)	Q(1), R(1), Q(2), R(2)
(G(K), K = 1, 7, 3)	G(1), G(4), G(7)
((A(I, J), I = 3, 5), J = 1, 9, 4)	A(3,1), A(4,1), A(5,1), A(3,5), A(4,5), A(5,5), A(3,9), A(4,9), A(5,9)
(R(M), M = 1, 2), I, ZAP(3), (R(3), T(I), I = 1, 3)	R(1), R(2), I, ZAP(3), R(3), T(1), R(3), T(2), R(3), T(3)

Thus, the elements of a square matrix, for example, may be transmitted in an order different from the order in which they appear in storage.

The array A(3,3) occupies storage in the order A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3).

By specifying the transmission of the array with the DO-implied list item

((A(I, J), J = 1, 3), I = 1, 3)

the transmission will be

A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3), A(3,1), A(3,2), A(3,3)

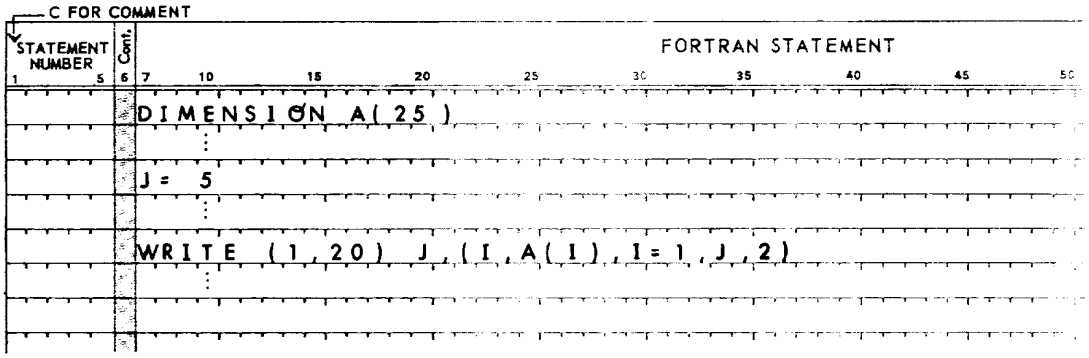
Special List Considerations

1. The ordering of a list is from left to right with repetition of items enclosed in parentheses (other than subscripts) when accompanied by controlling DO-implied indexing parameters.
2. An unsubscripted array name in a list implies the entire array.
3. Constants may appear in input/output lists only as subscripts or as indexing parameters.
4. For input lists the DO-implying index parameters (i, m_1, m_2, m_3) may not appear within the parentheses as list items.

   Examples:

READ(1,20)(I, J, A(I), I = 1, J, 2)	is not allowed.
READ(1,20) I, J, (A(I), I = 1, J, 2)	is allowed.
WRITE (1,20)(I, J, A(I), I = 1, J, 2)	is allowed.

Consider the following:



Output would be:

5, 1, A(1), 3, A(3), 5, A(5)

5. The number of items in a single list is limited only by the statement length restrictions.

6. In a formatted transmission each item must have the correct format as specified by a FORMAT statement.

FORMAT Statements

FORMAT statements are used in conjunction with formatted READ and WRITE statements to specify data conversion methods and/or editing of data as it is transmitted between computer storage and external devices.

These statements are nonexecutable and must have statement labels for reference by input/output statements.

FORMAT statements have the form

FORMAT ($S_1, S_2, S_3, \ldots, S_n$)

where

$n \geq 0$

S_i is either a field descriptor of one of the forms described in the paragraphs below or a repeated group of such field descriptors in the form

$r(S_1, S_2, S_3, \ldots, S_m)$

where

$m \geq 0$

r is a repeat count as described below

S_j is a field descriptor

The word FORMAT and the parentheses must appear as shown.

Commas are field separators and must be present as shown or may be replaced by slashes or groups of slashes. (See "Field Separators".)

A field is defined as that part of an external record occupied by one transmission item.

Field Descriptors

Field descriptors describe the size of data fields and specify the type of conversion and editing to be exercised upon each transmitted item.

Field descriptors may be in any of the following forms:

Descriptor	Classification
rFw.d rEw.d rIw	Numeric Conversion
nX	Spacing Specifications
nHh rAw rZw	Hollerith (Alphanumeric) Transmission

where

w and n are nonzero integer constants that define the field width (including digits, decimal points, and algebraic signs) in the external data representation.

d is an integer specifying the number of fractional digits appearing in the external data representation.

F, E, and I indicate the type of conversion to be applied to the items in the input/output list.

H and X specify character data represented completely within the format descriptor. Such specifications have no corresponding items in input/output lists.

A is used to read or write alphanumeric data.

Z is used to read or write hexadecimal data.

r is an optional, nonzero integer indicating that the descriptor will be active during the transmission of up to r data.

h is a string of n alphanumeric characters.

The basic forms of these descriptors are described in the following paragraphs (i.e., without the optional r). The r specification is discussed in a separate paragraph.

I-Type Conversion

Form:

Iw

Only integer data may be converted by this type of conversion. w specifies field width.

Output. Internal values are converted to a string of decimal digits. Negative values are preceded by a minus sign. If the converted value does not fill the specified field, the digits are right justified in the field and preceded by blanks. If the string of converted digits exceeds the field width, only the least significant w characters are output. This is not treated as an error.

Examples:

Format Descriptor	Internal Value	Output (b indicates blanks)
I6	+281	bbb281
I6	-17631	-17631
I3	126	126
I3	-126	126
I3	46931	931

Input. A field of w characters is input and converted to internal integer format. A minus sign (-) may precede the integer digits. If a sign is not present, the value is considered positive.

Integer values in the range -32767 to +32767 are accepted. Blanks may appear anywhere in the field. Embedded and trailing blanks are treated as zeros.

Examples:

Format Descriptor	Input (b indicates blanks)	Internal Value
I4	b124	+124
I4	-124	-124
I7	bbb7631	7631
I6	-b1024	-1024
I5	-31024	-3102

F-Type Conversion

Form:

Fw.d

Real type data are processed using this conversion; w characters are processed, of which d are considered fractional.

Output. Internal values are converted and output as minus sign or blank (if positive), followed by the integer portion of the number, a decimal point, and d digits of the fractional portion of the number, rounded and truncated if necessary.

The converted characters are right justified in the field, w, with preceding blanks to fill the field if necessary. If the conversion produces more than w characters, only the rightmost w characters are output. This is not treated as an error. The relationship $w \geq d + 2 + n$, where n is the number of integer digits, must hold true to prevent loss of digits.

Examples:

Format Descriptor	Internal Value	Output (b indicates blanks)
F10.4	368.42	bb368.4200
F7.1	-4786.361	-4786.4
F8.4	.0375	bbb.0375
F6.4	4739.76	9.7600
F7.3	15.0	b15.000

Input. (See description under Input for E-Type Conversion.)

E-Type Conversion

Form:

Ew.d

Real type data are processed using this conversion; w characters are processed, of which d are considered fractional.

Output. Internal format values are converted, rounded to d digits, and output in the order given below as

1. A minus sign or blank (if positive)
2. A decimal point
3. d decimal digits
4. The letter E
5. The sign of the exponent (minus or blank)
6. Two exponent digits

The values, as described above, are right justified in the field, w, with preceding blanks to fill the field if necessary. If the conversion produces more than w characters, only the rightmost w characters are output. This is not treated as an error. To insure against this loss of characters on the left, the relationship $w \geq d + 6$ must be satisfied by the format descriptor.

Examples:

Format Descriptors	Internal Value	Output (b indicates blanks)
E12.5	76.573	bb.76573Eb02
E12.7	-32672.354	.3267235Eb05
E7.3	156.93	157Eb03
E13.4	-0.0012321	bbb-.1232E-02
E8.2	-3.567	-.36Eb01

Input. Input data to be processed for input under E or F conversions can be in a relatively loose format in the external medium. w characters are input for each value.

The format is identical for either conversion and is a combination of the following:

1. Leading blanks (ignored)
2. A + or - sign (an unsigned input is assumed positive)
3. A string of digits
4. A decimal point
5. A second string of digits
6. The character E
7. A + or - sign
8. A decimal exponent

Each item in the list above is optional; but, if format item 8 is present, 6 or 7 or both are required.

Embedded or trailing blanks are treated as zeros.

The input data must fall within the ranges specified for real type data.

Note in the following examples that if no decimal point is given among the input characters, the d in the format descriptor establishes the decimal point in conjunction with the exponent, if given. If a decimal point is included in the input characters, the d specification is ignored.

Examples:

Format Descriptor	Input (b indicates blanks	Internal Value
E10.3	+0.13756+4	+1375.60
E10.3	bbbbb17631	17.631
F8.3	b1628911	+1628.911
F8.3	b1.62891	+1.62891
F8.3	1.72E+02	+172.0
E7.1	-36.273	-36.273
F10.3	-763267E-3	-0.763267

The X Descriptor

Form:

nX

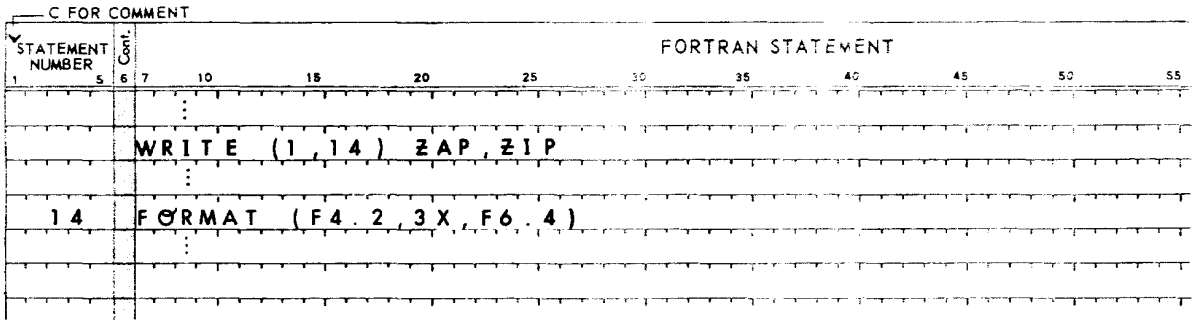
This descriptor causes no conversion to occur, nor does it correspond to items in an input/output list.

When used for output, nX causes n blanks to be inserted in the output record.

Under input conditions, processing nX causes n characters of the input record to be skipped.

Output Example:

If the variable ZAP contains 5.3 and the variable ZIP contains 7.6328, the statements



would output the record

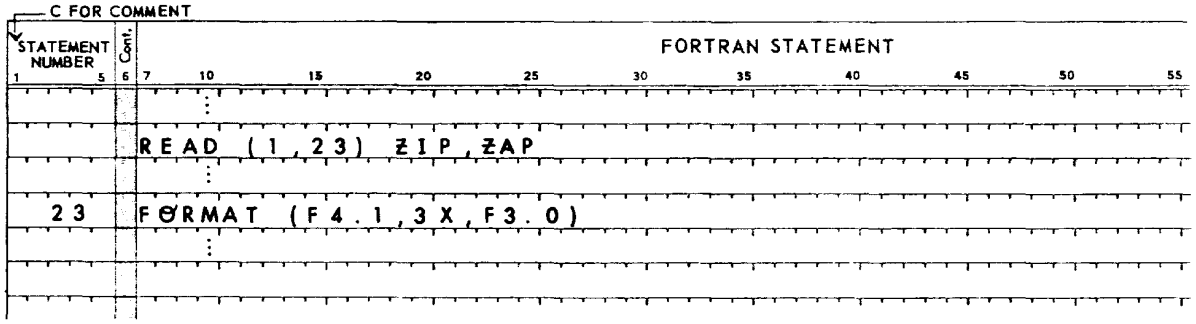
5.30bbb7.6328

Input Example:

If an external record contained

12.5ABC120

and the following statements are executed



the variable ZAP will contain 12.5 and ZIP will contain 120. ABC is skipped.

Hollerith (Alphanumeric) Transmission

Hollerith information may be transmitted by means of the H or A descriptor.

The H Descriptor

Form:

nHh

This descriptor causes Hollerith information to be read into, or written from, the n characters (h) following the nH descriptor in the format specification itself.

Output. The n characters (h) are output to the specified external device. Care should be taken that the character string h contains exactly n characters, so that the desired external field will be created, and so that subsequent data will be processed correctly. Blanks are counted as characters.

Examples:

Descriptor	Output (b = blanks)
1HR	R
8HbSTRINGb	bSTRINGb
12HXb(1,3) = 12.0	Xb(1,3) = 12.0

Input. The n characters of the string h are replaced by the next n characters of the input record. This results in a new string of characters in the field descriptor. Blanks are counted as characters.

Examples: (b = blanks)

Format Descriptor	Input Characters	Resultant Descriptor
5H12345	ABCDE	5HABCDE
8HbFALSEbb	bFALSEbb	8HbFALSEbb
9Hbbbbbbbbb	bMATRIXbb	9HbMATRIXbb

The A Descriptor

Form:

rAw

The A format code is used to read or write alphanumeric data. If w is equal to the number of characters corresponding to the length specification of the items in the I/O list, w characters are read or written.

Output. Internal binary values are converted to character strings at the rate of eight binary digits (two hexadecimal digits) per character. The most significant digits are converted first; that is, conversion is from left to right. The number of characters produced by an item depends on the number of words of storage allocated for that type of item. Normally, alphanumeric information is used with integer variables.

When the magnitude of w does not provide for enough positions to express the data completely, the external field is shortened from the right (least significant) end. This is not treated as an error condition. When w has a value greater than necessary, the external character string is right justified in the field and preceded by the appropriate number of blank characters.

Input. When the width w is larger than necessary (that is, when its magnitude is greater than the number of characters associated with the data type of the corresponding list item), the number of characters equal to the difference between w and the length specification are skipped and the remaining characters are read.

When the value of w is less than the number of characters associated with the data type of the list item, the most significant positions of the list item are filled with w characters, and the remainder of the positions are filled with blanks.

Hexadecimal Transmission

Hexadecimal information may be transmitted by means of the Z descriptor.

Z Descriptor (Hexadecimal)

Form:

rZw

The Z format code is used to read or write hexadecimal data. If w is equal to the number of hexadecimal digits corresponding to the length specification of the items in the I/O list, w characters are read or written.

Output. Internal binary values are converted to character strings at the rate of four binary digits (one hexadecimal digit) per character. The most significant digits are converted first; that is, conversion is from left to right. The number of characters produced by an item depends on the number of words of storage allocated for that type of item. Normally, hexadecimal information is used with integer variables.

When the magnitude of w does not provide for enough positions to express the data completely, the external field is shortened from the left (most significant end). This is not treated as an error condition. When w has a value greater than necessary, the external character string is right justified in the field and preceded by the appropriate number of blank characters.

Input. When the width w is larger than necessary (that is, when its magnitude is greater than the number of hexadecimal digits associated with the data type of the corresponding list item), the number of characters equal to the difference between w and the number of hexadecimal digits in the data item are skipped and the remaining characters are read.

When the value of w is less than the number of hexadecimal digits associated with the data type of the list item, the least significant positions of the list item are filled with w hexadecimal digits, and the remainder of the positions are filled with zeros.

Descriptor Repeat Specifications

Repeated use of a descriptor or descriptors is accomplished in one of three ways:

1. The E, F, and I descriptors may be immediately preceded by a repeat count, r, in the forms rEw.d, rFw.d, and rIw.

   In these cases, if the input/output list warrants it, the descriptor will be interpreted repetitively up to r times. r must be an integer constant.

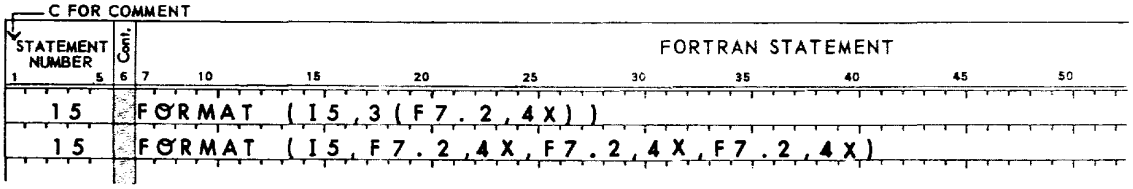
   The following examples show equivalent FORMAT statements:

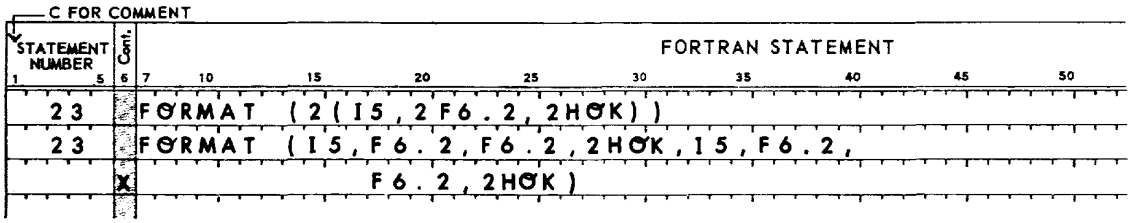
```
C FOR COMMENT
STATEMENT NUMBER  Cont.  FORTRAN STATEMENT
1        5        6     7  10  15  20  25  30  35  40  45  50
3                       FORMAT (4F7.3,F13.6)
3                       FORMAT (F7.3,F7.3,F7.3,F7.3,F13.6)
```

```
C FOR COMMENT
STATEMENT NUMBER  Cont.  FORTRAN STATEMENT
1        5        6     7  10  15  20  25  30  35  40  45  50
72                      FORMAT (2I6,3I4,2E12.3)
72                      FORMAT (I6,I6,I4,I4,I4,E12.3,E12.3)
```

2. Repetitive interpretation of a group of descriptors is accomplished by enclosing the group in parentheses and optionally preceding the left parentheses with an integer constant, called the group repeat count, which indicates the number of times the group is to be interpreted. If a group count is absent, a count of 1 is assumed. Only one level of parentheses is permitted within the standard format specification parentheses.

Note the following similar statements:

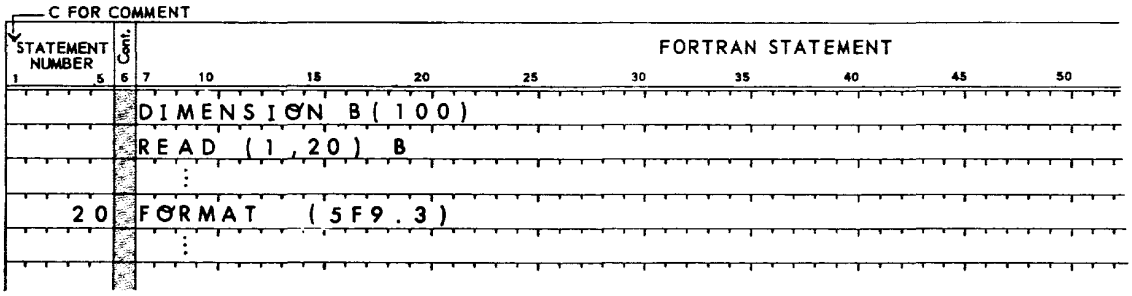




3. Repetition of format descriptor interpretation is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list to be processed.

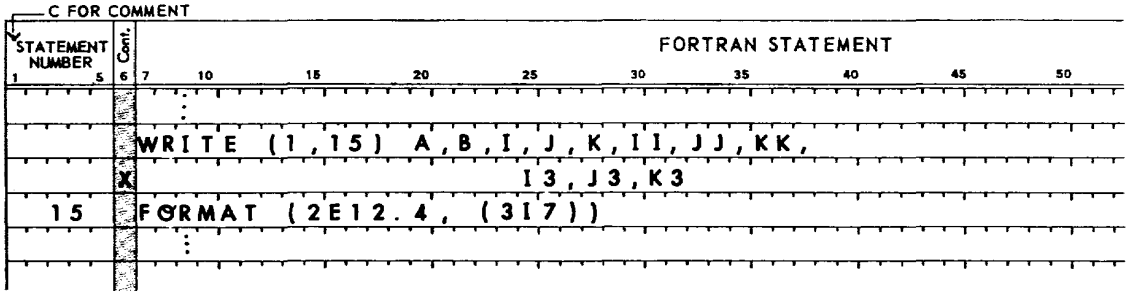
   When format control has proceeded to the last outer right parenthesis and another list item is specified, format control demands that a new record start; and, control reverts to that group repeat specification terminated by the last preceding right parenthesis or, if none exists, to the first left parenthesis of the format specification.

   Input Example:



   In this example, the descriptor 5F9.3 is used 20 times. The first 5 quantities from each of 20 records are input and assigned to array B.

   Output Example:



   In this example, 3 records are output. Record 1 contains A, B, I, J, and K. Because the descriptor (3I7) is reused twice, record 2 contains II, JJ, and KK. Record 3 contains I3, J3, and K3.

Field Separators

An H-format field is terminated after the specified number of characters; thus a field separator following an H-format field is optional. In all other cases, when two or more field descriptors are included in a format specification, they must be separated by ~~one or more slashes~~ either a comma or slash(es).

Example:

3X2HOK,F6.3 is incorrect

3X2HOKF6.3 is acceptable

A slash(/) separator, or a series of slashes, may be used instead of a comma. A slash not only separates field descriptors, but also specifies the demarcation of formatted records.

Each slash terminates a record and sets up the next record for processing. The remainder of an input record is ignored; the remainder of a fixed length output record is filled with blanks. Successive slashes (///.../) cause successive records to be ignored on input and successive blank records to be written on output. If a slash or group of slashes is used, commas must not be used concurrently.

Example:

(F5.4,/,4F10.3) is erroneous

(F5.4/4F10.3) is correct

Output Example:

```
C FOR COMMENT
STATEMENT NUMBER | Cont. | FORTRAN STATEMENT
                 |       | DIMENSION A(30), J(20)
                 |       | WRITE (1,7) (J(I),I=1,20),(A(I),I=1,30)
               7 |       | FORMAT (10I7/ 10I7/ 15F7.3/ 15F7.3)
```

In this example, the data specified by the list of the WRITE statement are output to unit 1 according to specifications of FORMAT statement 7. Four records are written as follows:

RECORD 1: J(1), J(2), . . . , J(10)

RECORD 2: J(11), J(12), . . . , J(20)

RECORD 3: A(1), A(2), . . . , A(15)

RECORD 4: A(16), A(17), . . . , A(30)

Input Example:

```
C FOR COMMENT
STATEMENT NUMBER | Cont. | FORTRAN STATEMENT
                 |       | DIMENSION BETA(10)
                 |       | READ (5,20) BETA(1),BETA(2),(BETA(I),I=3,10)
              20 |       | FORMAT (F10.2/ F10.2/ / 8F10.2)
```

In this example, the two array elements BETA(1) and BETA(2) receive their values from the first data fields of successive records (the remainders of the two records are ignored). The third record is ignored, and the remaining elements of the array are filled from the fourth record.

Relationships between FORMAT Control, List Specifications, and Record Demarcation

The following relationships and interactions between FORMAT control, input/output lists, and record demarcation should be noted.

1. Execution of a formatted READ or WRITE statement initiates FORMAT control.

2. The conversion performed on data depends on information jointly provided by the elements in the input/output list and field descriptors in the FORMAT statement.

3. If there is an input/output list, at least one descriptor of types E, F, or I must be present in the FORMAT statement.

4. Each execution of a formatted READ statement causes a new record to be input.

5. Each item in a FORMAT control input list corresponds to a string of characters in the record and to a descriptor of the types E, F, or I in the FORMAT statement.

6. H and X descriptors communicate information directly between the external record and the field descriptors without reference to list items.

7. On input, whenever a slash is encountered in the FORMAT statement, the entire record has been processed, or the FORMAT descriptors have been exhausted and reuse of the descriptors is initiated, processing of the current record is terminated, and the following occurs:

   a. Any unprocessed characters in the record are ignored.

   b. If more input is necessary to satisfy list requirements, the READ is reinitiated to process the next record.

8. A READ statement is terminated when all items in the input list have been satisfied if:

   a. The next FORMAT descriptor is E, F, or I, or

   b. The format control has reached the last outer right parenthesis of the FORMAT statement.

   If the input list has been satisfied, but the next FORMAT descriptor is H, X, or slash, more data are processed (with the possibility of new records being input) until one of the above conditions exists.

9. If format control reaches the last right parenthesis of the FORMAT statement but there are more list items to be processed, all or part of the descriptors will be reused.

10. When a formatted WRITE statement is executed and a slash is encountered in the FORMAT statement, format control has reached the rightmost right parenthesis, or 132 characters have been output, processing of the current record is terminated and if more output is necessary to satisfy list requirements, output of a new record is initiated.

Auxiliary I/O Statements

Three auxiliary I/O statements are provided.

REWIND Statement

The REWIND statement is normally used to rewind a file assigned to a magnetic tape unit. If the file is a direct access file, the associated variable for the file will be reset to the value 1.

The form of the statement is

```
REWIND u
```

where

u is an integer variable or constant designating the file unit number.

BACKSPACE Statement

The BACKSPACE statement is normally used to backspace one logical record in a file assigned to a magnetic tape unit. If the file is a direct access file, the associated variable is decremented by 1, subject to the constraint that it may never be decremented to less than 1.

The form of the statement is

```
BACKSPACE u
```

where

u is an integer variable or constant designating the file unit number.

ENDFILE Statement

The ENDFILE statement is normally used to write a file mark on a file assigned to a magnetic tape unit.

The form of the statement is

```
ENDFILE u
```

where

u is an integer variable or constant designating the file unit number.

Carriage Control for Printed Output

The first character position in an output record that is intended for printing may control the printer carriage by containing certain characters.

Vertical Format Control Characters

VFC	Printer Output	Typewriter Output	Other Output
blank	Print, single space	Print, new line	No effect
1	Eject page, print, single space	New line, print, new line	No effect
0	Single space, print, single space	New line, print, new line	No effect
-	Print	Print, new line	No effect
other	Print, single space	Print, new line	No effect

Note: On printer and typewriter output with VFC defined by a 0, 1, or -, a blank is output in place of the VFC in the first character position. The buffer containing the record is not changed, however.

The general effect of the vertical format control characters is:

1	Page eject
0	Double space
-	Overprint
other	Single space

Figure 2 is an example of an unformatted program utilizing auxiliary I/O statements.

Figure 3 is an example of a program using formatted I/O statements.

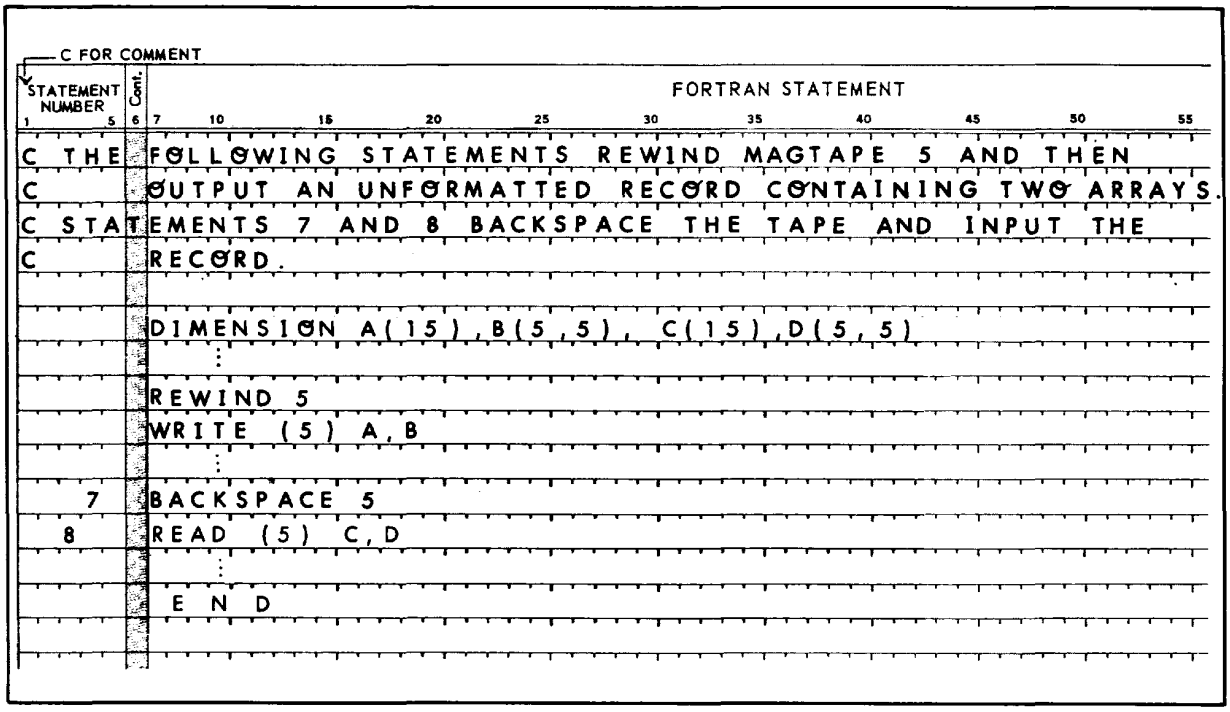


Figure 2. Input/Output Example 1 – Unformatted I/O

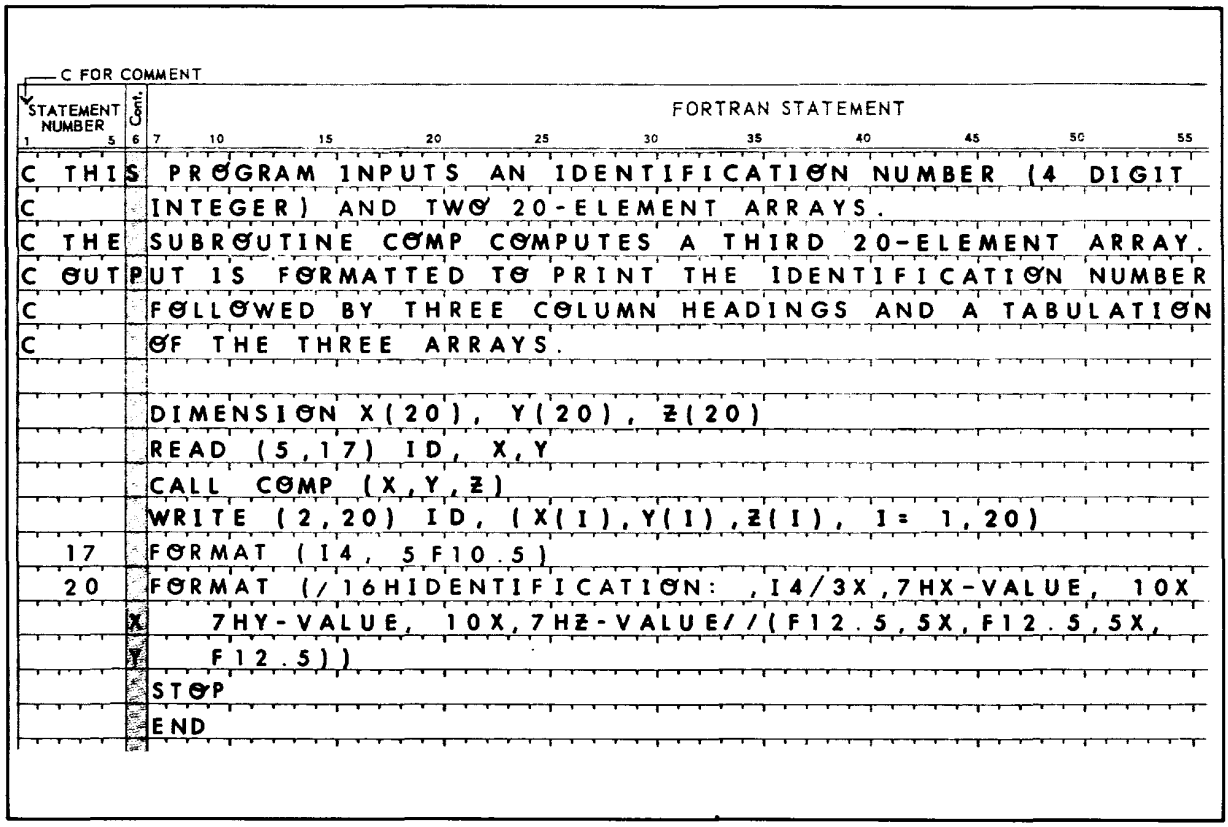


Figure 3. Input/Output Example 2 – Formatted I/O

⊙ Direct Access Input/Output (Basic FORTRAN IV only)

⊙ Direct Access Input/Output Statements

There are three direct access Input/Output Statements

1. READ
2. WRITE
3. DEFINE FILE

These statements identify a location within a data set from which data is read or written. As in sequential input/output, the FORMAT statement is sometimes used to specify the format of transmitted data.

Direct access READ and WRITE statements require definition of data sets by the DEFINE FILE statement and cannot act upon such sets without this definition. Definition of a data set by the DEFINE FILE statement must occur before any direct access READ or WRITE statement referencing the data set is executed.

DEFINE FILE has the form:

DEFINE FILE $n_1(r_1, m_1, a_1, i_1), n_2(r_2, m_2, a_2, i_2), \ldots, n_k(r_k, m_k, a_k, i_k)$

where

- n is an integer constant identifying the data set.
- r is an integer constant specifying the number of records in data set n.
- m is an integer constant specifying the maximum size of each record in data set n. Allowable record size increments are: storage locations, characters, or storage units. Record size determination is a function of "a" (see below).
- a designates the record size increment as follows:
  - L data set maximum record size specified as storage locations.
  - E data set maximum record size specified as characters.
  - U data set maximum record size specified as storage units (FORTRAN will assume 2 words/unit for standard precision and 3 words/unit for extended precision).
- i is the associated variable. It must be in COMMON or be local to the main program. The variable i is a nonsubscripted integer variable whose value is the record number immediately following the last record transmitted at the end of a read/write operation.

Example:

```
DEFINE FILE 3(40,90,E,K4)
```

This data set has the identifying number 3. It contains 40 records with a maximum length of 90 characters. K4 contains the number of the next record to be transmitted.

The same data could be defined 3(40,30,U,K4) if in extended precision mode, or 3(40,45,L,K4) if in standard precision mode.

More than one data set may be defined by the DEFINE FILE statement by placing a comma between parameter sets. (E.g., DEFINE FILE 5(25,60,L,J7), 8(75,75,E,M3), etc.)

⊙ Programming Considerations

Direct access I/O programming entails a relationship between FORTRAN records and the records described by the DEFINE FILE statement. In formatted input/output, all of the FORMAT statement conditions apply. For example, in a data set such as

```
DEFINE FILE 4(15,40,L,L9)
```

the controlling FORMAT statement could not specify a record exceeding 80 characters; e.g., FORMAT (5E16.1) would be acceptable, but FORMAT (I10,3F25.1) would not be acceptable.

In unformatted input/output the size of the transmitted record may not exceed the record size designated by the appropriate DEFINE FILE statement.

Assume the I/O list of a WRITE statement designates 24 real values; acceptable DEFINE FILE statements would be:

```
DEFINE FILE 14(30, 72, L, M7)
DEFINE FILE 43(20, 24, U, M7)
```

An unacceptable statement would be

```
DEFINE FILE 6(30, 3, L, M7)
```

If direct access I/O is to be used from a foreground program, the data set must be a permanent file that has been assigned to the specified data set number. The file must have been previously defined through the use of the RAD Editor, and the file definition must be the same as would normally be defined by the given DEFINE FILE statement.

READ Statement

The READ statement causes the transfer of data from direct access external devices to core memory.

The forms of the statement are

```
READ (d'p, n, ERR=b) list
READ (d'p, ERR=b) list
```

where

- d is the data set reference number expressed as an integer constant or an unsigned integer variable. Note the apostrophe, which is mandatory, following d.
- p is the record number in the data set d. It is an integer expression.
- n is the FORMAT statement number, if this is a formatted READ statement.
- b is the statement number of the statement to which control is transferred in the event of an unrecoverable input/output error. The error return (,ERR = b) is optional.
- list is the same as an I/O list for sequential input/output.

Example:

```
      DIMENSION J(1000)
      DEFINE FILE 9(100,100,E,MUD), 84(25,35,L,MUL)
15    READ (84'2) ALPHA, BETA, GAMMA
      .
      .
      .
23    FORMAT (4I10)
26    READ (9'50,23) (J(L),L = 1,4)
      .
      .
      .
      END
```

READ statement 15 reads data from the second record of data set 84 into ALPHA, BETA, and GAMMA. MUL contains 3 after execution of the READ statement.

READ statement 26 reads from record 50 of data set 9 under control of FORMAT statement 23. Integers are read into J(1), J(2), J(3), and J(4). Subsequent to the execution of READ statement 26, the variable MUD contains 51.

The FORMAT statement may be used to control reading; e.g., if the FORMAT statement in the example was

```
23 FORMAT (////28I25)
```

records 50-53 would have been skipped and record 54 read. MUD would contain 55.

WRITE Statement

The WRITE statement causes the transfer of data from core memory to an external direct access device.

The forms of the statement are

```
WRITE (d'p,n) list
WRITE (d'p) list
```

where

d is the data set reference number expressed as an integer constant or an unsigned integer variable. Note the apostrophe, which is mandatory, following d.

p is the record number in the data set d. It is an integer expression.

n is the FORMAT statement number, if the statement is formatted.

list is the same as an I/O list for sequential input/output.

9. SUBPROGRAMS

FORTRAN provides means for defining and/or using subprograms such that they may be referenced and brought into the logical execution sequence of a program unit wherever and as often as needed.

These subprograms may be part of a source language program, a separately compiled series of statements, or part of the FORTRAN function library.

There are four categories of subprograms:

1. Statement functions
2. Library functions
3. FUNCTION subprograms
4. SUBROUTINE subprograms

The first three categories are referred to collectively as functions; the last as subroutines.

Certain features of structure, reference, and terminology are common to all subprogram categories or to two or more categories:

1. A subprogram is identified by an identifier (see Chapter 2).
2. Any function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas.
3. Function references may occur as elements in arithmetic expressions.
4. Functions are single valued (i.e., they return a single result to the program unit from which the function was referenced). The type of the value is dependent on the IJKLMN rule unless explicitly typed.
5. A FUNCTION or SUBROUTINE subprogram constitutes a program unit.
6. Other than statement functions, all subprograms are defined externally to the program unit that references them.

Statement Functions

A statement function is defined by a single statement similar in form to an arithmetic assignment statement and is relevant only to the program unit in which it appears.

Defining Statement Functions

A statement function is defined by a statement of the form

$$f(u_1, u_2, \ldots, u_3) = e$$

where

- f is the function name
- u_i are dummy arguments
- e is an expression

Rules of Order and Structure

The order and structure of statement functions are governed by the following rules.

1. Statement functions, if they exist in a program unit, must precede all executable statements in the unit and must follow the specification statements.
2. If a statement function is referenced by another statement function, the referenced statement must precede the the referencing statement.

3. The name of a statement function, f, must not appear as a variable name or array name in the same program unit.
4. The u_i must be variable names, called the dummy arguments of the function. Array names, array element names, constants, and subprogram names are not allowed.
5. The u_i, being dummy arguments, serve only to indicate type, number, and order of arguments and may be the same as variable names of the same type appearing elsewhere in the program unit.
6. Aside from the dummy arguments, the expression e may contain only:
   a. Constants
   b. Variable references
   c. Library function references
   d. References to previously defined statement functions
   e. FUNCTION subprogram references
7. The relationship between f and e must conform to the assignment rules in Table 2, Chapter 5.
8. If the name of the function has appeared in a type statement, then the function value will be of such type. Otherwise, the IJKLMN rule will imply type.

Referencing Statement Functions

A statement function reference has the form

f(a)

where

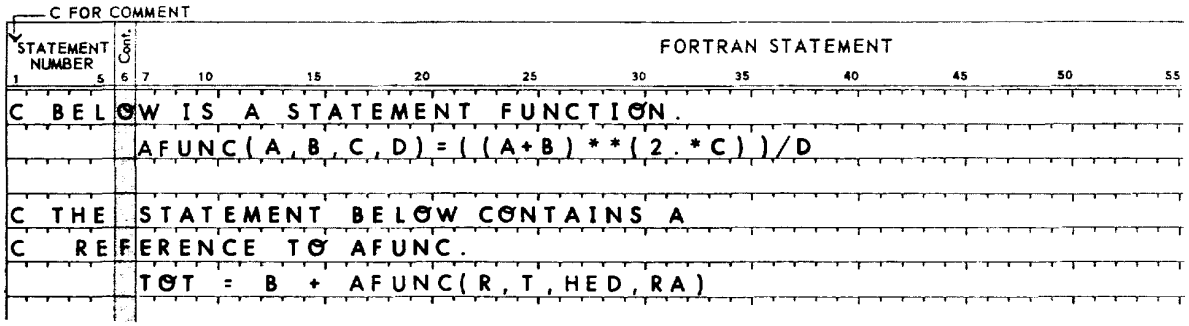
f is the identifier of the statement function

a is an actual argument list

Execution of a statement function reference results in the association of actual arguments with the corresponding dummy arguments in the expression of the function definition and an evaluation of the expression.

Following this, the resultant value is made available to the expression that contained the function reference.

Example:



Library Functions

The identifiers of library functions are predefined to the FORTRAN processor. However, use of these names to reference library functions in one program unit does not preclude use of the same names to identify other entities in different program units of the same executable program.

A library function reference is of the form

f(a)

where

f is the identifier of the library function

a is an actual argument list

The actual arguments must agree in type, number, and order with the library function specifications in Table 4 and may be any expression of the specified type.

Table 4. Library Functions

Function Name	Number of Arguments	Type of Argument	Type of Result	Definition of Function
ABS	1	Real	Real	Absolute value.
AINT	1	Real	Real	Integer part of argument expressed as a real value.
ALOG	1	Real	Real	Natural logarithm (base e).
ALOG10	1	Real	Real	Common logarithm (base 10).
AMAX0	N ≥ 2	Integer	Real	Maximum value for integer values.
AMAX1	N ≥ 2	Real	Real	Maximum value for real values.
AMIN0	N ≥ 2	Integer	Real	Minimum value for integer values.
AMIN1	N ≥ 2	Real	Real	Minimum value for real values.
AMOD	2	Real	Real	Arg_1 (mod arg_2). Evaluated as $arg_1 - arg_2 * AINT(arg_1/arg_2)$ i.e., the sign is the same as arg_1. Function undefined if $arg_2 = 0$.
ATAN	1	Real	Real	Arctangent in radians. Arg_1 = ordinate (y), arg_2 = abscissa (x). If arg_2 not present, assumed 1. Result (R) is arctangent of arg_1/arg_2 quadrant allocated in the range $-\pi < R \leq \pi$; ATAN(0,0) = 0.
ATAN2	2	Real	Real	
COS	1	Real	Real	Cosine of angle in radians.
COSH	1	Real	Real	Hyperbolic cosine.
DIM	2	Real	Real	Positive difference. DIM(x,y) = x -min(x,y).
EXP	1	Real	Real	Exponential (e**arg).
FLOAT	1	Integer	Real	Argument converted to a real value.
IABS	1	Integer	Integer	Integer absolute value.
IDIM	2	Integer	Integer	Integer positive difference. IDIM(j,k) = j - MIN(j,k).
INT	1	Real	Integer	Argument converted to an integer value.
IFIX	1	Real	Integer	

Table 4. Library Functions (cont.)

Function Name	Number of Arguments	Type of Argument	Type of Result	Definition of Function
ISIGN	2	Integer	Integer	Integer magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
MAX0	N ≥ 2	Integer	Integer	Integer maximum value.
MAX1	N ≥ 2	Real	Integer	Integer maximum value.
MIN0	N ≥ 2	Integer	Integer	Integer minimum value.
MIN 1	N ≥ 2	Real	Integer	Integer minimum value.
MOD	2	Integer	Integer	Arg_1 (mod arg_2). Evaluated as $arg_1 - arg_2 * [arg_1/arg_2]$ where the bracket indicates integer part; i.e., the sign is the same as arg_1. Function is undefined if $arg_2 = 0$.
SIGN	2	Real	Real	Magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
SIN	1	Real	Real	Sine of angle in radians.
SINH	1	Real	Real	Hyperbolic sine.
SQRT	1	Real	Real	Square root (positive value).
TANH	1	Real	Real	Hyperbolic tangent.

Execution of a library function reference results in the actions indicated in Table 4, based on the value of the actual arguments. Following execution, the resultant value of the function is made available to the expression that contained the function reference.

Arguments for which the results of these functions are not mathematically defined or are of a data type other than that specified in Table 4 are improper.

Examples:

```
M = J + IABS(N)
K = IFIX(ALPHA)
AVG = A/FLOAT(K)
A(3) = B(I) - SQRT(A(2))
A = Y - SIN(Z)
```

FUNCTION Subprograms

A FUNCTION subprogram is a program unit constructed of a series of FORTRAN statements headed by a FUNCTION statement and followed by an END line. A FUNCTION statement has one of the forms

```
FUNCTION f(υ1, υ2 ..., υn)
REAL FUNCTION f(υ1, υ2, ..., υn)
INTEGER FUNCTION f(υ1, υ2, ..., υn)
```

where

f is the identifier of the function to be defined

υ_i are dummy arguments that represent the variable or array names

Example:

```
FUNCTION BAL(A, B, I)
```

In this example BAL is the identifier of the function subprogram, and A, B, and I are dummy arguments. The function type is real.

FUNCTION Subprogram Construction

Construction of FUNCTION subprograms must comply with the following rules.

1. A FUNCTION statement must be the first statement of the subprogram.
2. The identifier of the function must appear as a variable name in the subprogram. During each execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in the subprogram is called the value of the function.
3. The value of the function is of the type explicitly stated (INTEGER or REAL); otherwise, it adheres to the IJKLMN rule.
4. The value of a function may be defined by the appearance of the function name on the left side of the equal sign of an arithmetic assignment statement or as an item in the list of an input statement.
5. The identifier of the function must not appear in any nonexecutable statement in the program unit other than in the FUNCTION statement.
6. The identifiers of dummy arguments can not appear in an EQUIVALENCE or COMMON statement in the subprogram.
7. A function subprogram cannot redefine any of its arguments or any entities in COMMON.
8. A function subprogram may contain any statements other than SUBROUTINE or another FUNCTION statement.
9. A function subprogram must contain at least one RETURN statement. A RETURN statement is the logical termination of the subprogram.
10. An END line must be the last line of a function subprogram to signify the physical end of a subprogram.

Examples are shown in Figures 4 and 5.

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–55)
C THIS		FUNCTION SUBPROGRAM COMPUTES THE DIFFERENCE
C		BETWEEN THE LARGEST AND SMALLEST VALUED ELEMENTS
C		OF A SINGLE DIMENSION ARRAY OF 100 ELEMENTS.
		FUNTION DIFF (ARY)
		DIMENSION ARY (100)
		S = ARY(1)
		G = S
		DO 23 I= 2,100
		IF (S-ARY(I)) 20,20,19
19		S = ARY(I)
		GO TO 23
20		IF (G-ARY(I)) 21,23,23
21		G = ARY(I)
23		CONTINUE
C THE		FOLLOWING STATEMENT DEFINES THE VALUE OF THE
C		FUNCTION.
		DIFF = G-S
		RETURN
		END

Figure 4. FUNCTION Subprogram Example 1 – DIFF

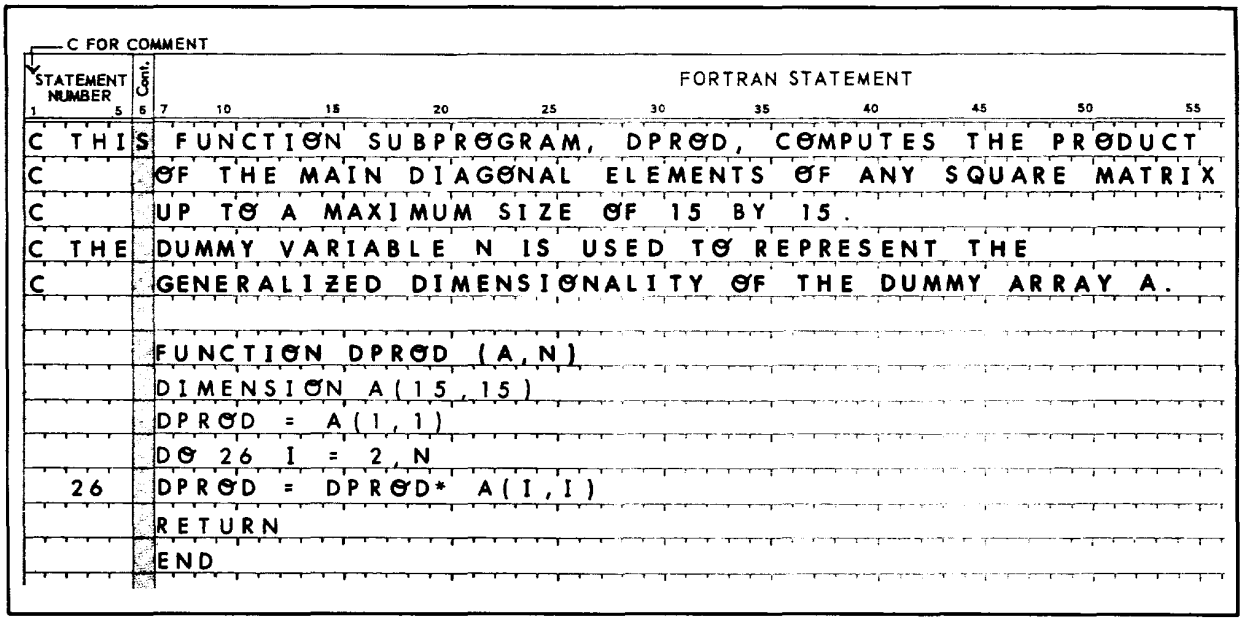


Figure 5. FUNCTION Subprogram Example 2 – DPROD

Referencing FUNCTION Subprograms

A FUNCTION subprogram reference is of the form

f(a)

where

f is the identifier of the function

a is an actual argument list

The actual arguments that constitute the argument list must agree in order, number, and type with the dummy arguments in the function defining program unit. An actual argument may be any one of the following:

1. A variable name
2. An array element name
3. An array name
4. Any other expression

Execution of a FUNCTION subprogram reference causes the actual arguments of the reference to be associated with all appearances of corresponding dummy arguments in executable statements and statement function definition statements within the subprogram.

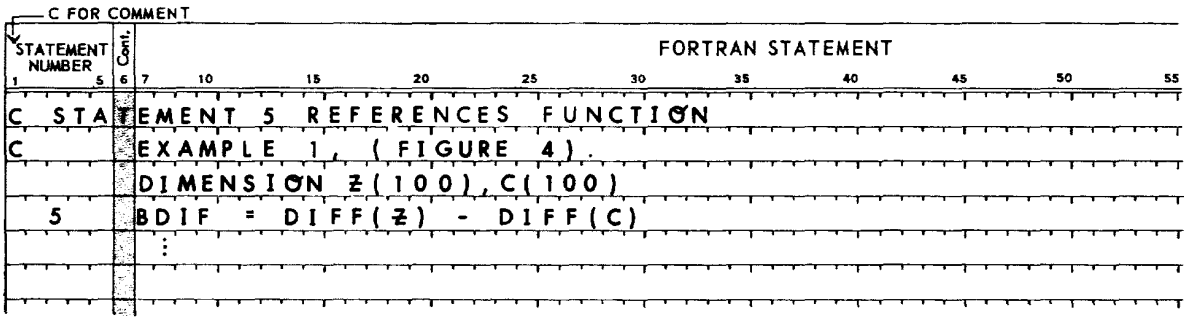
If an actual argument is an expression as specified in item 4 above, the expression is evaluated and the association is by value rather than by name.

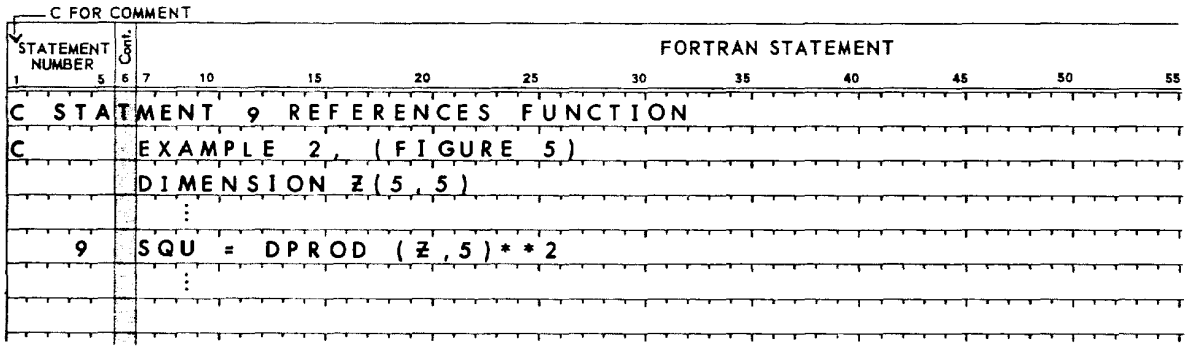
Following these associations, execution of the first executable statement of the subprogram is begun.

Array elements used as actual arguments may contain variables in their subscripts. Such subscripts are evaluated when the function reference is executed, and the determined value is used as the subscript as if the array had a constant subscript.

If a dummy argument of a FUNCTION subprogram is an array name, the corresponding actual argument must be an array name.

Examples:





SUBROUTINE Subprograms

A SUBROUTINE subprogram is a program unit constructed of a series of FORTRAN statements headed by a SUBROUTINE statement and followed by an END line.

A SUBROUTINE statement has either of the forms

SUBROUTINE $s(a_1, a_2, \ldots, a_n)$

SUBROUTINE s

where

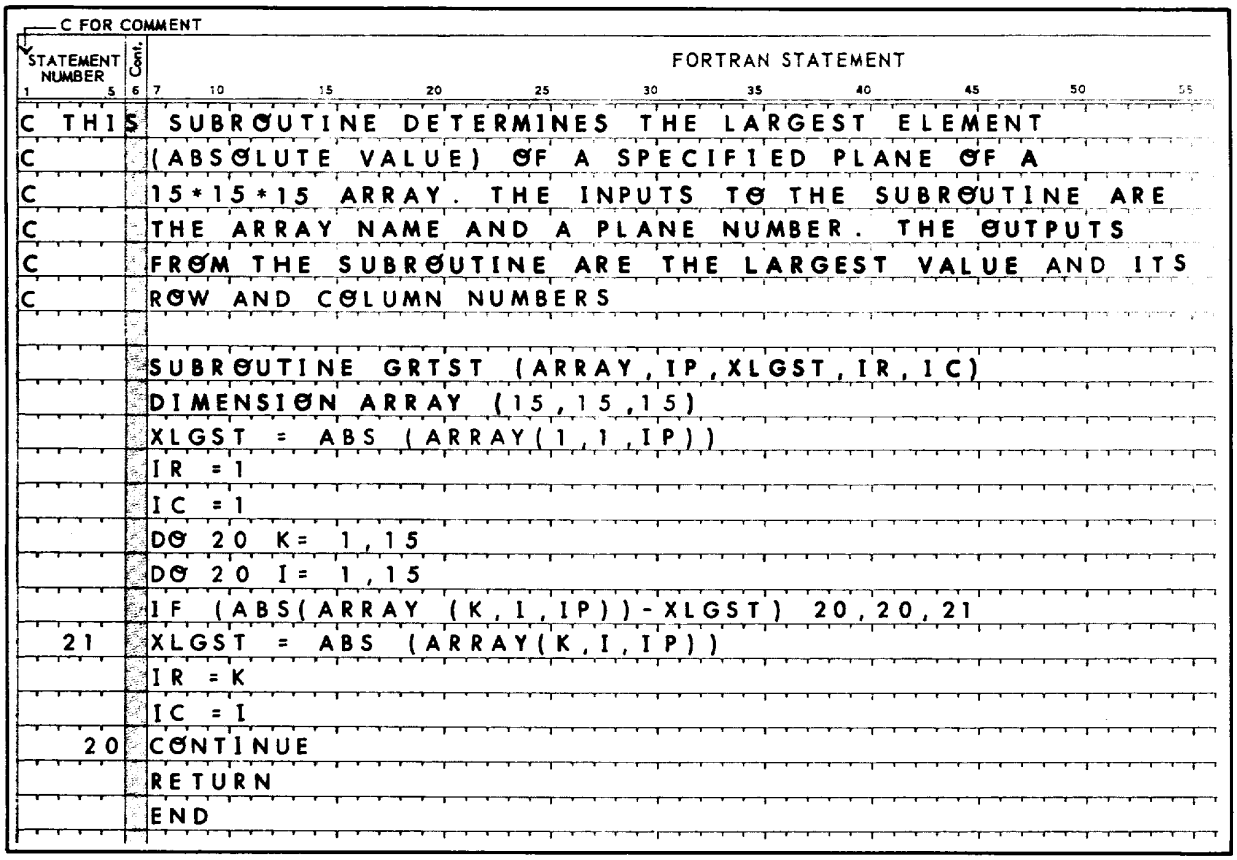
s is the identifier of the subroutine

a_i are dummy arguments that represent variable or array names

SUBROUTINE Subprogram Construction

Construction of SUBROUTINE subprograms must comply with the following rules.

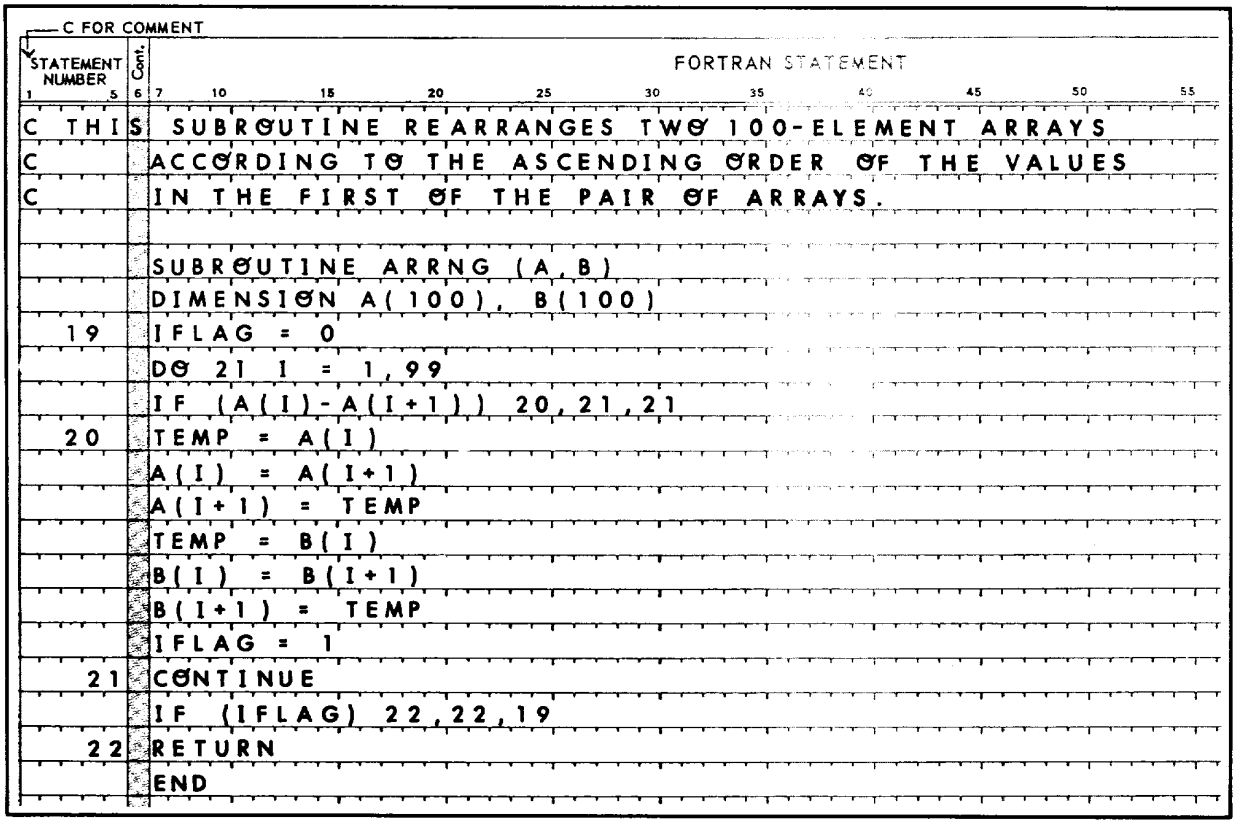
1. A SUBROUTINE statement must be the first statement of the subprogram.
2. The identifier of a subroutine must not appear in any statement in the subprogram except in the SUBROUTINE statement itself.
3. Dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the subprogram.
4. A subroutine subprogram may contain any statements except FUNCTION or another SUBROUTINE statement.
5. A SUBROUTINE subprogram must contain at least one RETURN statement. A RETURN statement marks a logical terminal point in the program.
6. An END line must be the last line of a subroutine subprogram to signify the physical end of the subprogram.



```
C FOR COMMENT
STATEMENT NUMBER (1-5)  Cont. (6)  FORTRAN STATEMENT (7-55)
C THIS  SUBROUTINE DETERMINES THE LARGEST ELEMENT
C       (ABSOLUTE VALUE) OF A SPECIFIED PLANE OF A
C       15*15*15 ARRAY. THE INPUTS TO THE SUBROUTINE ARE
C       THE ARRAY NAME AND A PLANE NUMBER. THE OUTPUTS
C       FROM THE SUBROUTINE ARE THE LARGEST VALUE AND ITS
C       ROW AND COLUMN NUMBERS

        SUBROUTINE GRTST (ARRAY,IP,XLGST,IR,IC)
        DIMENSION ARRAY (15,15,15)
        XLGST = ABS (ARRAY(1,1,IP))
        IR = 1
        IC = 1
        DO 20 K= 1,15
        DO 20 I= 1,15
        IF (ABS(ARRAY (K,I,IP))-XLGST) 20,20,21
   21   XLGST = ABS (ARRAY(K,I,IP))
        IR = K
        IC = I
   20   CONTINUE
        RETURN
        END
```

Figure 6. SUBROUTINE Subprogram Example 1 – GRTST



```
C FOR COMMENT
STATEMENT NUMBER (1-5)  Cont. (6)  FORTRAN STATEMENT (7-55)
C THIS  SUBROUTINE REARRANGES TWO 100-ELEMENT ARRAYS
C       ACCORDING TO THE ASCENDING ORDER OF THE VALUES
C       IN THE FIRST OF THE PAIR OF ARRAYS.

        SUBROUTINE ARRNG (A,B)
        DIMENSION A(100), B(100)
   19   IFLAG = 0
        DO 21 I = 1,99
        IF (A(I)-A(I+1)) 20,21,21
   20   TEMP = A(I)
        A(I) = A(I+1)
        A(I+1) = TEMP
        TEMP = B(I)
        B(I) = B(I+1)
        B(I+1) = TEMP
        IFLAG = 1
   21   CONTINUE
        IF (IFLAG) 22,22,19
   22   RETURN
        END
```

Figure 7. SUBROUTINE Subprogram Example 2 – ARRNG

Referencing Subroutines

A subroutine is referenced by a CALL statement (discussed in Chapter 7). The actual arguments that appear in the CALL statement argument list must agree in order, number, and type with the corresponding dummy arguments in the subroutine being called.

The actual argument in a subroutine reference may be one of the following:

1. A variable name
2. An array element name
3. An array name
4. Any other expression

Execution of a subroutine reference causes the actual arguments of the subroutine reference, if any, to be associated with the appearance of corresponding dummy arguments in executable statements or function definition statements within the subroutine.

Only arguments of the first three types above may be associated with dummy arguments that appear on the left of the equal sign of an arithmetic assignment statement or as an item in an input list. When such an association occurs, the values defined or redefined for these arguments are available to the calling program following execution of the subroutine.

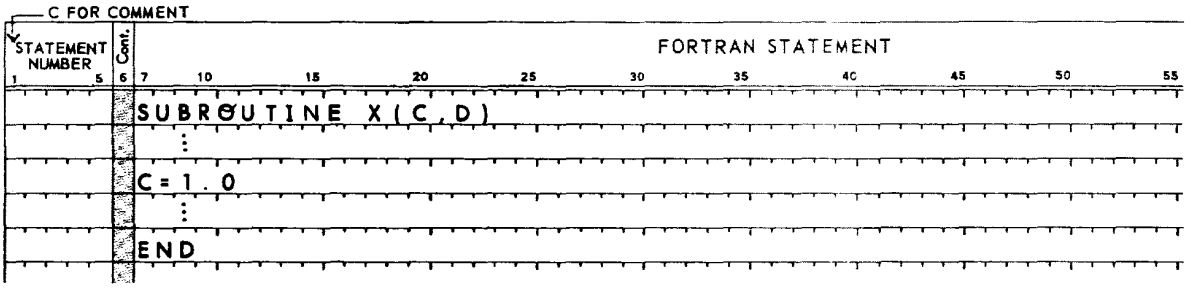
If any actual argument is as specified in item 4 above, the expression is evaluated and the association is by value rather than by name.

Following these associations, execution of the first executable statement is begun.

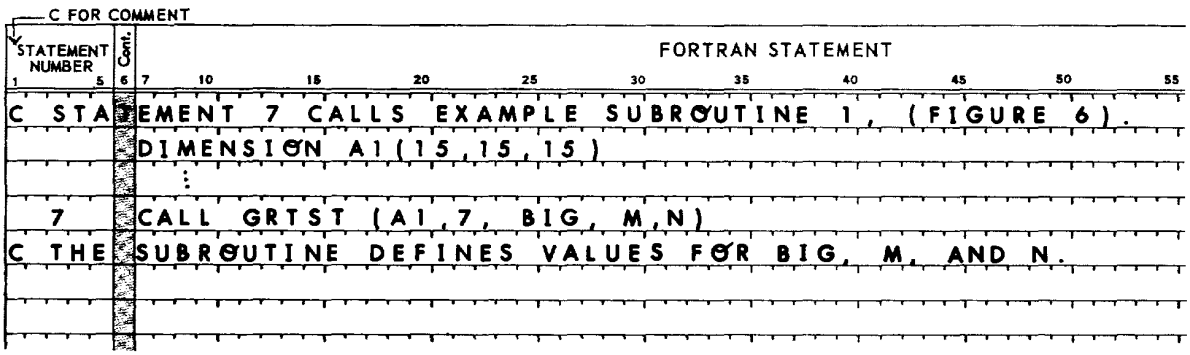
A subroutine reference may contain, as an actual argument, an array element name with variables in the subscript. Such subscripts are evaluated when the subroutine reference is executed, and the determined value is used as the subscript just as if the array had had a constant subscript.

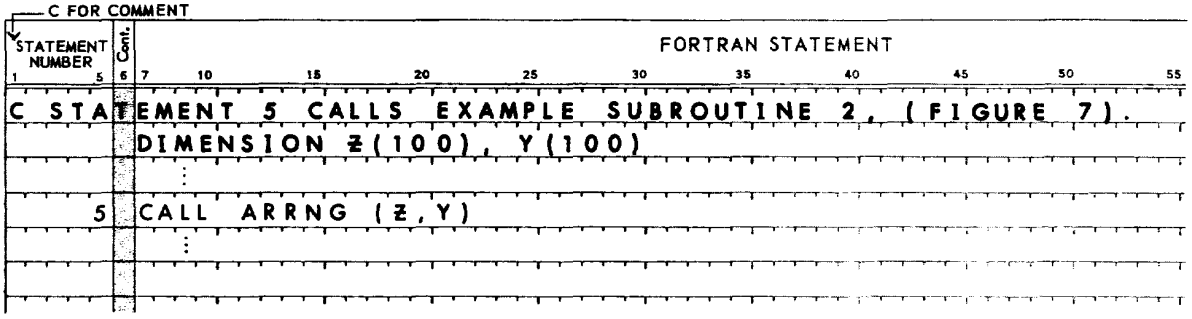
If a dummy argument of the subroutine is an array name, the corresponding actual argument must be an array name.

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine, or with an entity in COMMON, neither entity may be defined within the subroutine. For example, CALL X(A,A) is prohibited if the subroutine contains:



Examples:




C FOR COMMENT
STATEMENT NUMBER
Cont.
FORTRAN STATEMENT
1 5 6 7 10 15 20 25 30 35 40 45 50 55
C STATEMENT 5 CALLS EXAMPLE SUBROUTINE 2, (FIGURE 7).
DIMENSION Z(100), Y(100)
5 CALL ARRNG (Z,Y)

10. PROGRAMS AND PROGRAM COMPONENTS

A FORTRAN source program is a collection of FORTRAN statements, comment lines, and end lines that completely describe a computing procedure. Such programs are composed of any number of program units. Every program must contain one main program unit and may also contain one or more subprogram units.

A main program unit must contain at least one executable statement and must not contain a SUBROUTINE or FUNCTION statement.

There are two types of subprogram units:

1. SUBROUTINE subprograms
2. FUNCTION subprograms

The form and rules for construction of subprograms are discussed in Chapter 9.

Program Components

FORTRAN programs consist of program parts, program bodies, and subprogram statements.

The following definitions and explanations clarify the roles these components play in the structure of a program.

Program part. A program part must contain at least one executable statement, but need not contain FORMAT statements, and may not contain specification statements.

Program body. A program body is a collection of optional specification statements optionally followed by statement function definitions, followed by a program part followed by an END line. Specification statements must be in the order: DIMENSION, COMMON, EQUIVALENCE.

Subprogram. A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body.

Main Program. A main program consists of a program body.

Executable Program. An executable program consists of a main program plus any number of subprograms, external procedures, or both.

Program Unit. A program unit is a main program or a subprogram.

Program Execution Sequence

Execution of a program begins with the execution of the first executable statement of the main program. When a subprogram is referenced, execution of the subprogram begins with the first executable statement of that subprogram.

Completion of execution of a statement causes execution of the next following executable statement unless the statement being executed is a GO TO, Arithmetic IF, RETURN, or STOP statement, or the terminal statement of a DO. The sequence of execution following the execution of one of these statements is described in Chapter 7.

INDEX

F

G

H

I

L

M

N

U

V

W

X

Z

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well Organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, Please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your Name & Return Address

2190(12-72)

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

Fold

First Class
Permit No. 229
El Segundo,
California

BUSINESS REPLY MAIL

No postage stamp necessary if mailed in the United States

Postage will be paid by

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

Attn: Programming Publications

Fold