CONTROL DATA

1604/1604-A COMPUTER

1604/1604-A

FORTRAN 63/REFERENCE MANUAL

VOLUME 1 ARITHMETIC AND LOGICAL INFORMATION

PRELIMINARY

CONTROL DATA 1604/1604-A COMPUTER

FORTRAN 63/REFERENCE MANUAL

CONTROL DATA CORPORATION
8100 34th Avenue South
Minneapolis 20, Minnesota

March, 1963
Pub. No. 527

PREFACE

The FORTRAN*-63 language contains all of the features of its predecessor, FORTRAN-62 and forms an overset of the FORTRAN II language. The FORTRAN-63 compiler adapts current compiler techniques to the particular capabilities of the *Control Data* ** 1604 and 3600 computer systems. Emphasis has been placed on producing highly efficient object programs while maintaining the efficiency of compilation of FORTRAN-62.

This reference manual was written as a text for FORTRAN-63 classes and as a reference manual for programmers using the FORTRAN-63 system. The manual assumes a basic knowledge of the FORTRAN language.

Reference material consists of three volumes:

Volume I	Arithmetic and Logical Information
Volume II	Input - Output
Volume III	Will contain programming instructions for type non-standard arithmetic and instructions for compilation and execution of FORTRAN-63.

*FORTRAN is an abbreviation for FORmula TRANslation and was originally developed for International Business Machine equipment.

**Registered trademark of Control Data Corporation

CONTENTS

CONTENTS

ELEMENTS OF FORTRAN-63

1

This chapter presents the character set used by FORTRAN-63 and demonstrates how entities within it are used to form the identifier. Arithmetic, relational and logical operators are listed along with their meanings, and ideas relating to number are presented in terms of the fundamental definitions of quantity, variable and constant.

1.1 CHARACTERS

FORTRAN-63 uses the following character set; the conventional FORTRAN definitions apply:

The alphabet	A through Z
The Arabic numerals	0 through 9
The special characters	+ - = / ( ) . , $ * space

The special character $ is a statement separator which may be used to write more than one statement to a line.

1.2 OPERATORS

The operation symbols used in replacement and conditional statements are tabulated below:

Mathematical Symbol	Meaning	FORTRAN-63 Operators	Classification
+	Addition	+	Arithmetic
-	Subtraction	-	
÷	Division	/	
X	Multiplication	*	
[ ]^n	Exponentiation	**	
=	Equal to	.EQ.	Relational
≠	Not equal to	.NE.	
>	Greater than	.GT.	
≥	Greater than or equal to	.GE.	
<	Less than	.LT.	
≤	Less than or equal to	.LE.	
∧	Conjunction	.AND.	Logical
∨	Disjunction	.OR.	
¬	Negation	.NOT.	

Mathematical Symbol	Meaning	FORTRAN-63 Operators	Classification
∩	Logical product	.AND.	Masking
∪	Logical sum	.OR.	
~	Complement	.NOT.	
←	Is replaced by	=	Replacement

1.3 IDENTIFIERS

Identifiers in FORTRAN-63 fall into two classes: numeric and alphanumeric. The numeric identifiers are:

Bank designators in the 3600 system represented by a single octal digit.

Statement numbers represented by a number appearing in card columns or printer positions 1 through 5. This number is in the range $1 \leq N \leq 99999$.

Block COMMON identifiers.

The alphanumeric identifiers which name variables, arrays, subroutines, functions and the like may be from one to eight alphanumeric characters; the first of which must be alphabetic. Spaces in any identifier are squeezed out; $A_{\wedge\wedge}6$ is the same as A6.

Examples

A156	SINEX
ALPHA	HEGEMONY
G	LUX31Z
HERA	A1B2C3D4

1.4 QUANTITIES AND WORD STRUCTURE

FORTRAN-63 manipulates floating point or integer quantities. Floating point quantities have an exponent and a fractional part. The following classes of numbers are floating point quantities.

REAL — Exponent and sign 11 bits; fraction and sign 37 bits; range of number (in magnitude) $10^{-308} \leq N \leq 10^{308}$ and zero; precision approximately 11 decimal digits.

DOUBLE — Exponent and sign 11 bits; fraction and sign 85 bits; range of number (in magnitude) $10^{-308} \leq N \leq 10^{308}$ and zero; precision approximately 25 decimal digits.

COMPLEX — Two reals as defined above.

The following classes of numbers are integer quantities:

INTEGER — Represented by 48 bits, first bit is the sign; range of number (in magnitude) $0 \leq N \leq 2^{47} - 1$; precision is up to 15 decimal digits.

LOGICAL	1 in bit position 47 represents the value TRUE. 0 in bit position 47 represents the value FALSE.
HOLLERITH	Binary coded decimal (BCD) representation treated as an integer number.

A FORTRAN-63 program may contain any or all of these classes of numbers in the forms of constants, variables, elements of arrays, evaluated functions and so forth. Variables, arrays and functions are associated with types assigned by the programmer. The type of a constant is determined by its form.

1.5 CONSTANTS

To define constants let:

- n be a string of decimal digits
- s be a scalar with a maximum of three decimal digits
- o be a string of octal digits
- h be the length of a Hollerith field
- f be a Hollerith field
- R be a Real

1.5.1 INTEGER CONSTANTS

n denotes an integer whose range, precision, et cetera, is as defined above.

Examples

63	3647631
247	2
314159265	464646464

1.5.2 OCTAL CONSTANTS

Octal constants may consist of up to 16 octal digits. The form of this constant is oB.

Examples

7777777700000000B	2323232323232323B
77777700077777B	77B
7777777777777700B	same as .NOT. 77B (section 2.7)
	or -77B

1.5.3 HOLLERITH CONSTANTS

A Hollerith constant is a string of alphanumeric characters of variable length of the form hHf, where h is an unsigned decimal integer between 1 and 8 representing the length of the field f. Spaces are significant in the field f. When h is less than 8, the representation in the computer word is left-justified with BCD spaces filling the remainder of the word. An alternate form is hRf. When h is less than 8, the internal representation is right-justified with zero fill.

Examples

6HCOGITO	8RCDC 3600
4HERGO	8R **
3HSUM	1H)

1.5.4 FLOATING POINT CONSTANTS

REAL

Real numbers are represented by a string of up to ten digits. A real constant may be expressed with a decimal point or with a fraction and an exponent representing a power of ten. The forms of real constants are:

n.n n. .n $nE^{+}_{-}s$ $.nE^{+}_{-}s$ $n.nE^{+}_{-}s$ $n.E^{+}_{-}s$

The plus sign may be omitted for positive s. The range of s is $0 \leq s \leq 308$.

Examples

3.1415768	31.41592E-001
314.	.31415E01
.6749162	.31415E001
314159E-05	.31415E+01

DOUBLE

Double precision constants are represented by a string of up to 25 digits. A double precision constant has forms analagous to the forms of reals, with the E replaced by D. The forms are:

n.nD n.D .nD $nD^{+}_{-}s$ $n.nD^{+}_{-}s$ $n.D^{+}_{-}s$

The plus sign may be omitted for positive s; the range of s is $0 \leq s \leq 308$. The D designator must always appear.

Examples

3.14159265358979323846 26D	31415.D-004
3.1415D	3798675244301 11D+001
3.1415D0	
3141.598D-03	

COMPLEX

Complex constants are represented by a pair of reals enclosed in parentheses with the reals separated by a comma: (R_1, R_2). R_1 represents the real part of the complex number and R_2 represents the imaginary part.

Examples

The complex numbers 1. + 6.55i, -14.09 + .0001654i, 15. + 16.7i, and -i are represented in FORTRAN-63 as:

(1., 6.55)	(-14.09, 1.654E-004)
(15., 16.7)	(0., -1.)

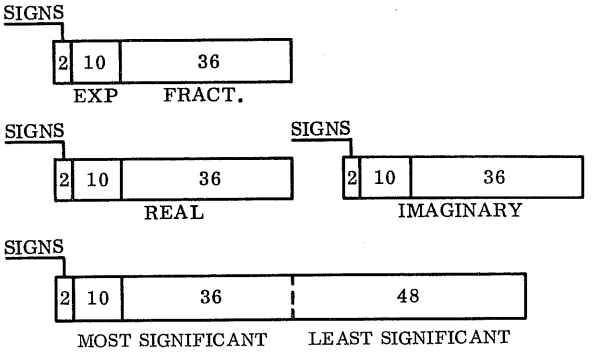
1.5.5

WORD STRUCTURE

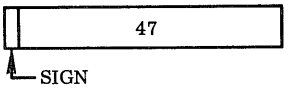
The word structure of the quantities in FORTRAN-63 is shown below:

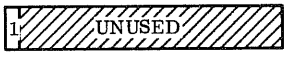
WORD STRUCTURE of STANDARD TYPES

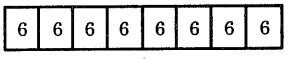
FLOATING POINT QUANTITIES



INTEGER QUANTITIES







TYPE	λ*	REMARKS
COMPLEX	2 WORDS	CONSECUTIVE MEMORY LOCATIONS
DOUBLE	2 WORDS	CONSECUTIVE MEMORY LOCATIONS
REAL	1 WORD	
INTEGER	1 WORD	
HOLLERITH	1 WORD	6 BITS / CHARACTER; APPENDIX B
LOGICAL	1 BIT	(FOR ARRAY STORAGE SEE SECTION 3.2)

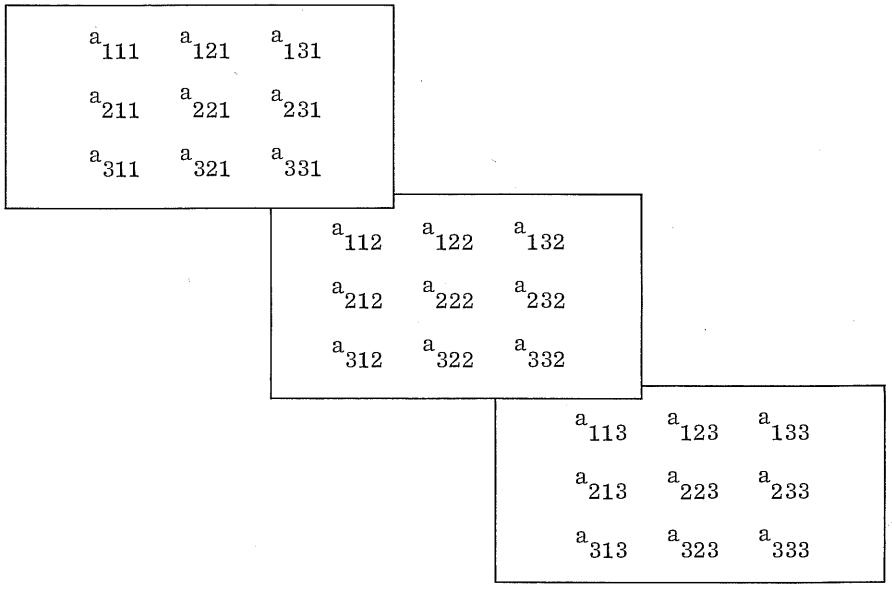
*ELEMENT LENGTH

1.6 VARIABLES

FORTRAN-63 recognizes two kinds of variables, each is represented by an alphanumeric identifier. A simple variable represents a single quantity; a subscripted variable represents a single quantity within an array of quantities. The identifier appears with a subscript list enclosed in parentheses. The subscript list has the form (S_1) or (S_1, S_2) or (S_1, S_2, S_3) where S_i may be standard or non-standard.

1.6.1 ARRAY STRUCTURE AND SUBSCRIPT FORMS

Elements of arrays are stored columnwise in ascending order of storage location. A 3 by 3 by 3 matrix illustrates the storing process:



The planes are stored in order, starting with the first, as follows:

$a_{111} \rightarrow L$ $a_{121} \rightarrow L+3$ $a_{133} \rightarrow L+24$

$a_{211} \rightarrow L+1$ $a_{221} \rightarrow L+4$ ... $a_{233} \rightarrow L+25$

$a_{311} \rightarrow L+2$ $a_{321} \rightarrow L+5$ $a_{333} \rightarrow L+26$

The maximum permissible number of subscripts appearing with a variable is three. The structure of the subscript is flexible within the classes standard and non-standard.

1.6.2 SUBSCRIPT RULES

SUB1 A standard subscript has one of the following forms:

C

C*m

$m \pm d$

m

$C*m \pm d$

where C, d are unsigned integer constants and m is a simple integer variable.

SUB2 A non-standard subscript is any arithmetic expression used as a subscript, or a subscripted subscript.

SUB3 The location of an array element with respect to the first element of the array is a function of the array dimension, type (3.1), and the subscripts appearing with the array identifier. In general, given DIMENSION A(L,M,N) the location of A (i,j,k) with respect to the first element A of the array is given by

$$A + \{i - 1 + L\,(j - 1 + M\,(k - 1))\}$$

The quantity in braces is called the subscript expression. For standard subscripts, the subscript expression, when evaluated, is an integer. For non-standard subscripts that are arithmetic expressions, the subscript expression is truncated after evaluation.

Examples

1. Referring to the matrix in 1.6.1, the location of A (2,2,3) with respect to A (1,1,1) is

   $$\text{Locn}\ \{A(2,2,3)\} = \text{Locn}\ \{A(1,1,1)\} + \{2-1+3(1+3(2))\}$$
   $$= L + 22$$

2. Given DIMENSION Z (5,5,5) and I = 1, K = 2, X = 45°, A =7.29, B = 1.62. The location, z, of Z (I * K, TANF (x), A −B) with respect to Z (1,1,1) is:

   z = Locn $\{Z(1,1,1)\}$ + $\{2-1+5(1-1+5(4.67))\}$ Integer part

   = Locn $\{Z(1,1,1)\}$ + $\{117.75\}$ Integer part

   = Locn $\{Z(1,1,1)\}$ + 117

SUB4 FORTRAN-63 permits the following relaxation on the representation of subscripted variables:

Given Array A declared with 3 dimensions as in DIMENSION $A(D_1,D_2,D_3)$ where the D_i are integer constants.

then

A(I,J,K)	implies A(I,J,K)
A(I,J)	implies A(I,J,1)
A(I)	implies A(I,1,1)
A	implies A(1,1,1)

similarly, for $A(D_1,D_2)$:

A(I,J)	implies A(I,J)
A(I)	implies A(I,1)
A	implies A(1,1)

The converse does not hold. The elements of a single-dimension array, $A(D_1)$, may not be referred to as A(I,J,K) or A(I,J).

Array allocation is discussed under DIMENSION, (3.2) and array structure (1.6.1).

Examples

Simple Variable	Subscripted Variable (Standard)	Subscripted Variable (Non Standard)
FRAN	A(I,J)	A(MAXF(I,J,M))
P	B(I+2,J+3,2*K+1)	B(J,SINF(J))
Z14	Q(14)	C(I+K)
ESTRUS	P(KLIM,JLIM+5)	MOTZO(3*K*ILIM+3.5)
MAX3	SAM(J-6)	WOW(I(J(K)))
I	B(1,2,3)	Q(1,-4,-2)

1.7 FORTRAN-63 LANGUAGE STATEMENTS

The FORTRAN-63 elements in this section are combined to form statements, the basic program element. These statements are either executable or non-executable. An executable statement performs a calculation or directs control of the program; a non-executable statement provides the compiler with information regarding variable structure, array allocation, storage sharing requirements, and so forth. FORTRAN-63 statements are listed in Appendix C.

EXPRESSIONS AND REPLACEMENT STATEMENTS

2

There are three kinds of expressions in FORTRAN-63: Arithmetic and masking † expressions which have numerical values and logical expressions which have truth values. These expressions may appear in arithmetic, masking or logical replacement statements, in IF statements or in arithmetic function statements.

2.1 ARITHMETIC EXPRESSIONS

All expressions are combinations of operators and operands. For the arithmetic expression, these entities are:

Operators:	** * / + -
Operands:	Constants
	Variables (Simple or subscripted)
	Functions (Chapter V)

Two or more arithmetic expressions can be combined to form another arithmetic expression and so on. Parentheses are used to direct the order of evaluation of the expression.

2.1.1 RULES OF FORMATION

A1 The hierarchy of arithmetic operations is:

**	exponentiation	class 1
/	division	class 2
*	multiplication	
+	addition	class 3
-	subtraction	

A2 Any variable (with or without subscripts) or constant, or function, is an arithmetic expression. These entities may be combined by using the arithmetic operators to form meaningful algebraic arithmetic expressions, subject to the following rules and definitions:

---

†Called Boolean in earlier versions of FORTRAN.

1. Let op be an arithmetic operator and X,Y be arithmetic expressions. The form X op op Y is never legitimate.

2. If X is an expression then (X), ((X)), et cetera, are expressions.

3. If X,Y are expressions, then

   X + Y
   X - Y
   X/Y
   X*Y are expressions.

4. Expressions of the form X**Y and X**(-Y) are legitimate. They are subject to the restrictions in Mode of Arithmetic Expressions (2.2.1).

5. Implied multiplication is permitted, but only in the following four ways:

constant(. . .)	implies	constant * (. . .)
(. . .) (. . .)	implies	(. . .) * (. . .)
(. . .)constant	implies	(. . .) * constant
(. . .)variable	implies	(. . .) * variable

A3 In an expression with no parentheses or within a pair of parentheses, in which unlike classes of operators appear, evaluation proceeds in the order stated in A1. In these expressions where operators of like classes appear, evaluation proceeds from left to right.

A4 When an arithmetic expression contains a function, the function is evaluated first.

A5 In parenthetical expressions within parenthetical expressions, evaluation begins with the innermost expression.

Examples

Expressions

```
A
3.141592
B + 16.8946
(A - B(I,J + K) )
G * C(J) + 4.1/  (Z(I+J,3*K) )*SINF(V)
(Q + V(M,MAXF(A,B) ) * Y**2) / (G*H-F(K + 3) )
-C + D(I,J) * 13.627
```

In the following examples R indicates an intermediate result in evaluation; it does not necessarily imply an object language store.

A**B/C+D*E*F-G is evaluated:

$A^{**}B \longrightarrow R_1$

$R_1/C \longrightarrow R_2$

$D^{*}E \longrightarrow R_3$

$R_3{}^{*}F \longrightarrow R_4$

$R_4+R_2 \longrightarrow R_5$

$R_5-G \longrightarrow R_6$ evaluation completed

A**B/(C+D)*(E*F-G) is evaluated:

$A^{**}B \longrightarrow R_1$

$C+D \longrightarrow R_2$

$E*F-G \longrightarrow R_3$

$R_1/R_2 \longrightarrow R_4$

$R_4 * R_3 \longrightarrow R_5$ evaluation completed

If the expression contains a function, the function is evaluated first.

H(13)+C(I,J+2)*(COSF(Z))**2 is evaluated:

$Z \longrightarrow R_1$

$COSF(R_1) \rightarrow R_2$

$R_2 * R_2 \longrightarrow R_3$

$R_3 * C(I,J+2) \rightarrow R_4$

$R_4 + H(13) \longrightarrow R_5$ evaluation completed

The following is an example of an expression with embedded parentheses.

A*(B+((C/D)-E)) is evaluated:

$C/D \longrightarrow R_1$

$R_1 - E \longrightarrow R_2$

$R_2 + B \longrightarrow R_3$

$R_3 * A \longrightarrow R_4$ evaluation completed

A*(SINF(X)+1.)-Z/(C*(D-(E+F))) is evaluated:

$SINF(X) \longrightarrow R_1$

$R_1 + 1. \longrightarrow R_2$

$R_2 * A \longrightarrow R_3$

$E+F \longrightarrow R_4$

$-R_4 \longrightarrow R_4$

$R_4 + D \longrightarrow R_5$

$R_5 * C \longrightarrow R_6$

$-Z \longrightarrow R_7$

$R_7/R_6 \longrightarrow R_8$

$R_8 + R_3 \longrightarrow R_9$ evaluation completed

2.1.2 INTEGER ARITHMETIC, CAUTION

In both the 1604 and 3600 computer systems, dividing an integer quantity by an integer quantity always yields a truncated (least integer) result; thus 11/3 = 3. For this reason, plus the fact that expressions containing operators of the same class are evaluated from left to right, the expression I*J/K is not necessarily the same as J/K*I. For example, 4*3/2 = 6 but 3/2*4 = 4.

2.2 MODE OF ARITHMETIC EXPRESSIONS

FORTRAN-63 permits full mixed mode arithmetic which increases flexibility in combining operand types. The five standard operand types are as follows:

COMPLEX	two words per element
DOUBLE	two words per element
REAL	one word per element
INTEGER	one word per element
LOGICAL	one bit per element

The programmer may define three non-standard types specifying multi-word elements or partial word elements, called bytes, whose length in bits is an integral divisor of 48. The mechanics of the TYPE declaration are covered in section 3.1.

Mixed mode arithmetic is completely general; however, most applications will probably mix operand types real and integer, real and double, or real and complex. The following rules establish the relationship between the mode of an evaluated expression and the types of the operands it contains.

2.2.1 TYPE AND MODE RULES

AM1 The order of dominance of the standard operand types within an expression from highest to lowest is:

COMPLEX
DOUBLE
REAL
INTEGER
LOGICAL

AM2 The mode of an evaluated arithmetic expression is referred to by the name of the dominant operand type.

AM3 In mixed arithmetic expressions containing non-standard types the following restrictions hold:

1. The non-standard types (types 5, 6, 7) may never be mixed with each other.
2. Any one of the types 5, 6, 7 may be mixed with any or all of the standard types. When this is done, the non-standard type dominates the hierarchy established in rule AM1.

AM4 In expressions of the form A**B the following rules apply:

1. Neither A nor B may be type logical or byte (non-standard) type.
2. B may be negative in which case the form is: A**(-B).
3. For the standard types (except logical) the mode/type relationships are:

Type A	Type B				
	I	R	D	C	
I	I	R	D	C	mode of A**B
R	R	R	D	C	
D	D	D	D	C	
C	C	C	C	C	

For example, if A is real and B is complex, the mode of A**B is complex.

4. If A or B or both are a non-standard multi-word type, the programmer must provide subroutines for the evaluation of A**B.

2.2.2 MIXED MODE EVALUATION

Examples

1. Given A, B type real; I, J type integer. The mode of expression A*B-I+J will be real because the dominant operand is type real. It is evaluated:

$A*B \longrightarrow R_1$	real	
Convert I to	real	
$R_1 - I \longrightarrow R_2$	real	
Convert J to	real	
$R_2 + J \longrightarrow R_3$	real	Evaluation completed

2. The use of parentheses may change the evaluation. A,B,I,J are defined as above. A*B-(I-J) is evaluated:

$I-J \longrightarrow R_1$	integer	
Convert R_1 to	real $\longrightarrow R_2$	
$A*B \longrightarrow R_3$	real	
$R_3 - R_2 \rightarrow R_4$	real	Evaluation completed

3. Given C1,C2 type complex; A1,A2 type real. The mode of expression A1*(C1/C2)+A2 will be complex because its dominant operand is type complex. It is evaluated:

$C1/C2 \rightarrow R_1$	complex	
Convert A1 to	complex	
$A1*R1 \rightarrow R_2$	complex	
Convert A2 to	complex	
$R_2+A2 \rightarrow R_3$	complex	Evaluation completed

4. Consider the expression C1/C2+(A1-A2) where the operands are defined as in 3 above. It is evaluated:

$A1-A2 \rightarrow R_1$	real	
Convert R_1 to	complex $\rightarrow R_2$	
$C1/C2 \rightarrow R_3$	complex	
$R_3 + R_2 \rightarrow R_4$	complex	Evaluation completed

Mixed mode arithmetic with standard types is illustrated by this example.

5. Given:
   - C complex
   - D double
   - R real
   - I integer
   - L logical

   and the expression C*D+R/I-L

The dominant operand type in this expression is type complex; therefore, the evaluated expression will be of mode complex. Evaluation:

Round D to a real and affix zero imaginary part

C*D $\longrightarrow R_1$	complex	
Convert R to	complex; convert I to complex	
R/I $\longrightarrow R_2$	complex	
$R_2+R_1 \rightarrow R_3$	complex	
Convert L to	complex	
$R_3-L \rightarrow R_4$	complex	Evaluation completed

If the same expression is rewritten with parentheses as C*D+(R/I-L) the evaluation proceeds:

Convert I to	real	
R/I $\longrightarrow R_1$	real	
Convert L to	real	
$R_1-L \rightarrow R_2$	real	
Convert R_2 to	complex $\rightarrow R_3$	
Round D to	real and affix zero imaginary part	
C*D $\longrightarrow R_4$	complex	
$R_4+R_3 \rightarrow R_5$	complex	Evaluation completed

2.3 MIXED MODE CONVERSIONS

Mixed mode arithmetic is accomplished through the special library subroutines. In the 1604 computer system, these routines include double precision and complex arithmetic. In the 3600 system, the double precision arithmetic is built into the hardware; the complex arithmetic is performed by a library subroutine.

2.4 ARITHMETIC REPLACEMENT STATEMENT

The general form of the arithmetic replacement statement (or simply, arithmetic statement) is A = F, where F is an arithmetic expression and A is an identifier representing a variable. The operator = means that the value of the evaluated expression, F, is assigned to A, with conversion for mode if necessary.

The identifier A is a variable; usually the type is a standard form: complex, double, real, integer, or logical.

Non-standard types may also be specified; they may be used as left-hand variables also.

Complex and double precision variables are floating point quantities requiring two computer words. Real, integer and logical variables are represented by one word. The mode of an evaluated expression is determined by the type of dominant operand. However, this does not restrict the types that A may assume. An expression of complex mode may replace A even if A is of type real. The following chart shows the A,F relationship for all the standard modes.

ARITHMETIC REPLACEMENT STATEMENT A=F

A is an Identifier F is an Arithmetic Expression

ϕ(f) is the Evaluated Arithmetic Expression

TYPE of A \ Mode of ϕ(f)	Complex	Double	Real	Integer
Complex	Store real & imaginary parts of ϕ(f) in real & imaginary parts of A.	Round ϕ(f) to real. Store in real part of A. Store zero in imaginary part of A.	Store ϕ(f) in real part of A. Store zero in imaginary part of A.	Convert ϕ(f) to real & store in real part of A. Store zero in imaginary part of A.
Double	Discard imaginary part of ϕ(f) & replace it with ±0 according to real part of ϕ(f).	Store ϕ(f) (most & least significant parts) in A (most & least significant parts).	If ϕ(f) is ± affix ±0 as least significant part. Store in A, most & least significant parts.	Convert ϕ(f) to real. If ϕ(f) is ±, affix ±0 as least significant part. Store in A, most & least significant parts.
Real	Store real part of ϕ(f) in A. Imaginary part is lost.	Round ϕ(f) to real & store in A. Least significant part of ϕ(f) is lost.	Store ϕ(f) in A.	Convert ϕ(f) to real. Store in A.
Integer	Convert real part of ϕ(f) to INTEGER. Store in A. Imaginary part is lost.	Round ϕ(f) to real, convert to INTEGER & store in A. The least significant part is lost.	Convert ϕ(f) to INTEGER. Store in A.	Store ϕ(f) in A.
Logical	If real part of ϕ(f)$\neq$0, 1→A. If real part of ϕ(f)=0, 0→A.	If ϕ(f)$\neq$0, store 1 in A. If ϕ(f)=0, store 0 in A.	Same as for double at left.	Same as for double at left.

When all of the operands in the expression F are of type logical, the expression is evaluated as if all the logical operands were integers. Let L_1, L_2, L_3, L_4 be logical variables, let R be a real variable and I an integer variable.

$I = L_1 * L_2 + L_3 - L_4$

will be evaluated as if the L_i were all integers (0 or 1) and the resulting value will be stored, as an integer, in I.

$R = L_1 * L_2 + L_3 - L_4$

is evaluated as stated above, but the result is converted to a real (a floating point quantity) before it is stored in R.

Examples

Given:

C_i ,	A_1	complex
D_i ,	A_2	double
R_i ,	A_3	real
I_i ,	A_4	integer
L_i ,	A_5	logical

1. $A_1 = C_1 * C_2 - C_3 / C_4$

   The mode of the expression is complex. Therefore the result of the expression is a two-word, floating point quantity. A_1 is type complex and the result replaces A_1.

2. $A_3 = C_1$

   The mode of the expression is complex. The type of A_3 is real; therefore the real part of C_1 replaces A_3.

3. $A_3 = C_1 * (0., -1.)$

   The mode of the expression is complex. The type of A_3 is real; the imaginary part of C_1 replaces A_3.

4. $A_4 = R_1 / R_2 * (R_3 - R_4) + I_1 - (I_2 * R_5)$

   The mode of the expression is real. The type of A_4 is integer; the result of the expression evaluation, a real, will be converted to an integer replacing A_4.

5. $A_2 = D_1 ** 2 * (D_2 + (D_3 * D_4)) + (D_2 * R_1 * R_2)$

   The mode of the expression is double. The type of A_2 is double; the result of the expression evaluation, a double precision floating quantity replaces A_2.

6. $A_5 = C_1 * R_1 - R_2 + I_1$

   The mode of the expression is complex. Since A_5 is type logical, an integer 1 will replace A_5 if the real part of the evaluated expression is not zero. If the real part is zero, zero replaces A_5.

2.5 LOGICAL AND RELATIONAL EXPRESSIONS

A logical expression has the general form

O_1 op O_2 op O_3 . . . op

where O_i are arithmetic expressions, relations, or variables of type logical, and op is one of the logical operators .NOT. .AND. .OR. The value of a

logical expression is either true or false. A / relational expression has the form $q_1 \rho q_2$ where at least one of q_1, q_2 is an arithmetic expression; the other q may be either an arithmetic expression or a single logical variable. ρ is an operator belonging to the set

.EQ. .NE. .GT. .GE. .LT. .LE.

A relation is true if q_1 stands in the relation ρ to q_2. A relation is false if q_1 does not stand in the relation ρ to q_2.

Within the compiler, relations are evaluated as illustrated in the following example. Consider the relation p = q.

This is equivalent to the question, does p-q = 0?

The compiler computes the difference and tests it for zero. If the difference is zero, the relation is true. If the difference is not zero, the relation is false.

Relational expressions are converted internally to arithmetic expressions according to the rules of mixed mode arithmetic. These expressions are evaluated and compared with zero to determine the truth value of the corresponding relational expression. When expressions of mode complex are tested for zero, only the real part is used in the comparison.

2.5.1 RULES GOVERNING RELATIONS

REL1 The only permissible forms of a relation are:

$q_1 \rho q_2$

q — by itself, in which case a non-zero value is true and a zero value is false.

REL2 $q_1 \rho q_2 \rho q_3$ . . is <u>not</u> permissible.

REL3 The evaluation of a relation of the form $q_1 \rho q_2$ is from left to right. The relations $q_1 \rho q_2$, $q_1 \rho (q_2)$, $(q_1) \rho q_2$, $(q_1) \rho (q_2)$ are equivalent.

Examples

A .GT. 16.	R(I) .GE. R(I-1)
R-Q(I)*Z .LE. 3.141592	K .LT. 16
B-C .NE. D+E	I .EQ. J(K)

2.5.2 LOGICAL EXPRESSION RULES

LOG1 The hierarchy of logical operations is:

First	.NOT.
then	.AND.
then	.OR.

LOG2 A logical variable or a relational expression is, in itself, a logical expression. If $\mathcal{L}_1$, $\mathcal{L}_2$ are logical expressions, then

.NOT. $\mathcal{L}_1$
$\mathcal{L}_1$ .AND. $\mathcal{L}_2$
$\mathcal{L}_1$ .OR. $\mathcal{L}_2$

are logical expressions. If $\mathcal{L}$ is a logical expression, $(\mathcal{L})$, $((\mathcal{L}))$ are logical expressions.

LOG3 If $\mathcal{L}_1, \mathcal{L}_2$ are logical expressions and op is .AND. or .OR. then, $\mathcal{L}_1$ op op $\mathcal{L}_2$ is never legitimate.

LOG4 .NOT. may appear in combination with .AND. or .OR. only as follows:

.AND..NOT.
.OR..NOT.
.AND.(.NOT. · · ·)
.OR.(.NOT. · · ·)

.NOT. may appear with itself only in the form .NOT.(.NOT.(.NOT. · · ·

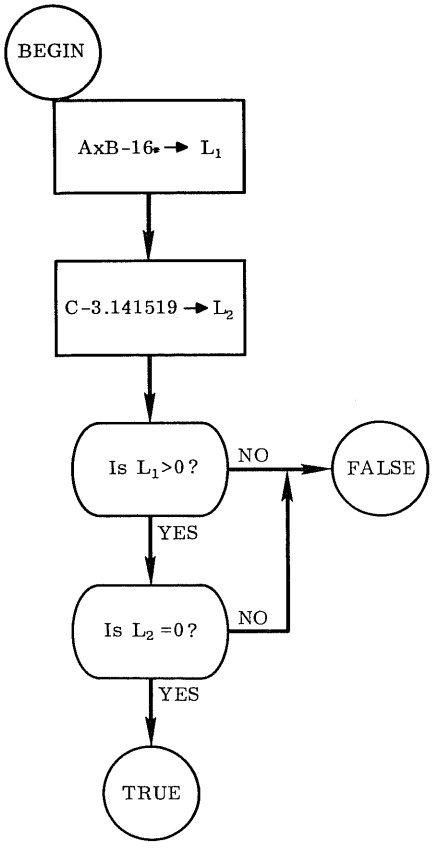
LOG5 If $\mathcal{L}_1$, $\mathcal{L}_2$ are logical expressions, the logical operators are defined as follows:

.NOT. $\mathcal{L}_1$	is false if $\mathcal{L}_1$ is true
$\mathcal{L}_1$ .AND. $\mathcal{L}_2$	is true if and only if $\mathcal{L}_1, \mathcal{L}_2$ are both true
$\mathcal{L}_1$ .OR. $\mathcal{L}_2$	is false if and only if $\mathcal{L}_1, \mathcal{L}_2$ are both false

Examples

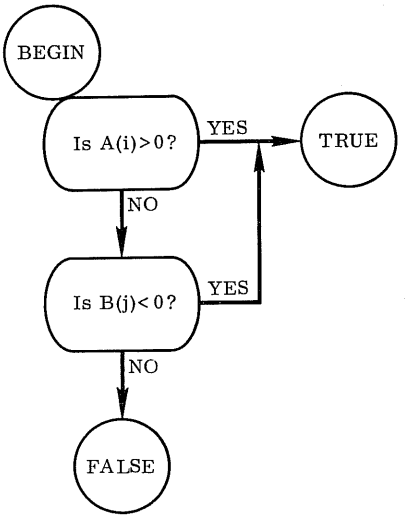
Logical Expressions

{The product A*B greater than 16.} .AND. {C equals 3.141519}
A*B .GT. 16. .AND. C .EQ. 3.141519



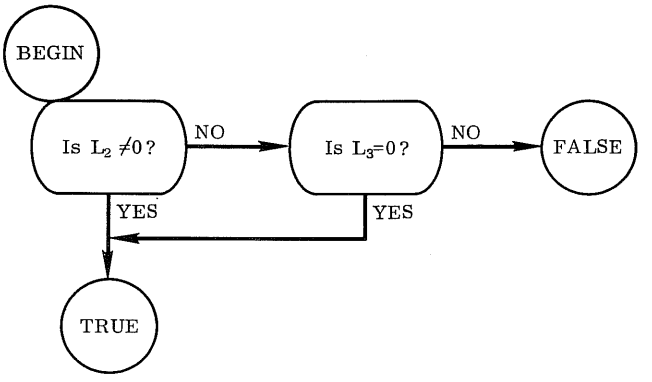
{A(I) greater than 0} .OR. {B(J) less than 0}

A(I) .GT. 0 .OR. B(J) .LT. 0

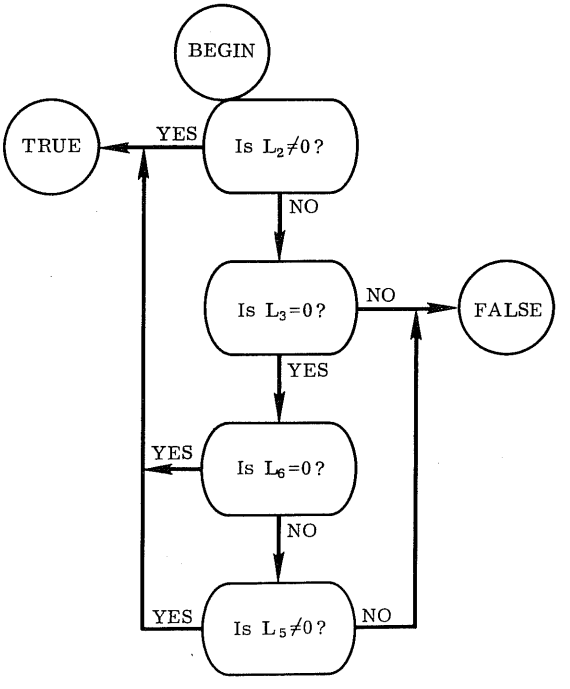


In the two examples below, all L_i are of TYPE LOGICAL

(L2 .OR. .NOT. L3)



L2 .OR. .NOT. L3 .AND. (.NOT. L6 .OR. L5)



Incorrect Usages

A.GT.(B.AND.C)
10.LE.N.LE.100
Q.NOT. .OR.R
C.AND. .NOT. .NOT.B

The last expression is permissible in the form C.AND. .NOT.(.NOT.B)

2.6 LOGICAL REPLACEMENT STATEMENT

The general form of the logical replacement statement is L=E, where L is a variable of type logical and E may be a logical expression or relation, or an arithmetic expression.

When an arithmetic expression appears in a logical replacement statement, that expression is examined for being zero or non-zero. If the expression is non-zero, the left hand variable, L has the value TRUE. If the expression is equal to zero, the left hand variable L has the value FALSE. Thus the treatment of arithmetic expressions in logical replacement statements is consistent with that given to logical expressions in logical replacement statements.

2.7 MASKING EXPRESSIONS

In FORTRAN-63, a masking expression is one in which 48-bit arithmetic is performed bit-by-bit on the operands within the expression. These operands must be of types real and integer only. Type integer includes octal and Hollerith constants.

The masking operators are: .NOT. .AND. .OR. Although these operators are identical in appearance to the logical operators, their meanings are different. For masking operators the following definitions apply:

.NOT. means complement the operand
.AND. means form the bit-by-bit logical product of two operands
.OR. means form the bit-by-bit logical sum of two operands

The operations are described below.

p	v	p .AND. v	p .OR. v	.NOT. p
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

2.7.1 MASKING EXPRESSION RULES

ME1 The hierarchy of operation is: first .NOT., then .AND., then .OR.

ME2 Let B_1, B_2 be variables or constants whose types are real or integer. Then the following are masking expressions.

.NOT. B_1

B_1 .AND. B_2

B_1 .OR. B_2

ME3 If B is a masking expression, then (B), ((B)), are masking expressions.

ME4 .NOT. may appear with .AND. or .OR. only as follows:

.AND..NOT.
.OR..NOT.
.AND. (.NOT. • • •)
.OR. (.NOT. • • •)

ME5 Masking expressions of the following forms are evaluated from left to right.

A .AND. B .AND. C . . .
A .OR. B .OR. C . . .

Examples

Given:

A_1	7777000000000000	octal constant
A_2	0000000077777777	octal constant
B	0000000000001763	octal form of integer constant
C	2004500000000000	octal form of real

.NOT. A_1	is	0000777777777777
A_1 .AND. C	is	2004000000000000
A_1 .AND. .NOT. C	is	5773000000000000
B .OR. .NOT. A_2	is	7777777700001763

2.8 MASKING REPLACEMENT STATEMENT

The general form of the masking replacement statement is E = M. The masking statement is distinguished from the logical statement in the following ways.

1. The type of E must be real or integer.
2. All operands in the expression M must be type real or integer. M may contain parenthetical arithmetic subexpressions whose mode is real or integer.

Examples

Given: All variables of type real or integer.

```
A(I)   = B .OR. .NOT. C(I+2,J*K)
B      = D .AND. Q
C(I,J) = .NOT. Z(K) .AND. (Q1 .OR. .NOT. Q2)
TEST   = CELESTE .AND. 7HECLIPSE
AB     = D .OR. (S + T)
```

2.9 MULTIPLE REPLACEMENT STATEMENTS

The multiple replacement statement is a generalization of the replacement statements discussed earlier in this chapter, and its form is:

$\psi_n = \psi_{n-1} = \ldots = \psi_2 = \psi_1 =$ expression

The expression may be arithmetic, logical or masking. The ψ_i are variables subject to the following restrictions:

Arithmetic or Logical Statement: ψ_1 = EXP

If EXP is logical or arithmetic, then

If the variable ψ_1 is type complex, double, real, or integer, then ψ_1 = EXP is an arithmetic statement.

If the variable ψ_1 is type logical, then ψ_1 = EXP is a logical statement.

Masking Statement: ψ_1 = EXP

If EXP is a masking expression, ψ_1 must be a type real or integer variable only.

The remaining n-1 ψ_i may be variables of any type, and the multiple replacement statement replaces each of the variables $\psi_2, \ldots, \psi_n$ with the value of ψ_1 in a manner analogous to that employed in mixed mode arithmetic statements.

Examples

Given:		
	A, B, C, D	real
	E, F	complex
	G, H	double
	I, J	integer
	K, L	logical

The numbers in the examples represent the evaluations of expressions.

I = A = 4.6	4.6 → A	
	4 → I	
A = I = 4.6	4 → I	
	4.0 → A	
I = A = E = (10.2,3.0)	10.2 → E	real
	3.0 → E	imaginary
	10.2 → A	
	10 → I	
F = A = I = E =(13.4,16.2)	13.4 → E	real
	16.2 → E	imaginary
	13 → I	
	13.0 → A	
	13.0 → F	real
	0.0 → F	imaginary
K = I = -14.6	-14 → I	
	1 → K	
I = K = -14.6	1 → K	
	1 → I	

TYPE DECLARATIONS AND STORAGE ALLOCATION

3

This chapter discusses how FORTRAN-63 allocates storage. The relation between word structure (TYPE) and array length (DIMENSION, COMMON) is explained. The methods for sharing storage (EQUIVALENCE) and the DATA statement is explained.

3.1 TYPE DECLARATIONS

The TYPE declaration provides the compiler with information regarding the structure of the identifiers that name variables (1.6) and functions (5.1). The discussion that follows describes how type information is passed to the compiler from source language statements.

There are five standard variable types (non-standard types are explained in Volume III). Identifiers are declared of a given type by one of the following declarative statements:

```
TYPE   COMPLEX   List
TYPE   DOUBLE    List
TYPE   REAL      List
TYPE   INTEGER   List
TYPE   LOGICAL   List
```

A list, as used here, is a string of identifiers, in which each identifier is separated from the succeeding one by a comma. Subscripts are not permitted. An example of a list is:

```
A,B1,CAT,D36F, EUPHORIA
```

The characteristics of the standard variable types are:

Type	Element Definition	Quantification
Complex	2 words/Element	Floating point
Double	2 words/Element	Floating point
Real	1 word /Element	Floating point
Integer	1 word /Element	Integer
Logical	1 bit /Element	Logical

3.1.1 TYPE DECLARATION RULES

TD1 The TYPE statement is non-executable and must precede the first executable statement in a given program.

TD2 If a variable is declared differently in two or more TYPE statements, its TYPE will be determined from the last TYPE statement in which it appears.

TD3 A variable not declared in a TYPE statement will be interpreted as TYPE REAL if the first letter of its identifier is A, . . . ,H or O, . . .Z. It will be interpreted as TYPE INTEGER if the first letter of the identifier is I, J, K, L, M, N.

Examples

```
TYPE    COMPLEX    A147, RIGGISH, AT1LL2
TYPE    DOUBLE     TEEPEE, B2BAZ
TYPE    REAL       EL, CAMINO, REAL, IDE63
TYPE    INTEGER    QUID, PRO, QUO
TYPE    LOGICAL    GEORGE6
```

3.2 DIMENSION

A subscripted variable represents an element of an array of variables. Storage may be reserved for arrays by the non-executable statements DIMENSION or COMMON.

The standard form of the DIMENSION statement is:

DIMENSION V_1 ,V_2 , . . .

V_i have the form: Identifier (subscript string). The subscript string may have up to 3 unsigned constants separated by commas, as in SPACE(5,5,5). Under certain conditions within subprograms only, the subscripts may be integer variables. (Variable Dimensions 5.8)

The number of computer words reserved for a given array is a function of the product of the subscripts in the subscript string, and the type of the variable. In the statements

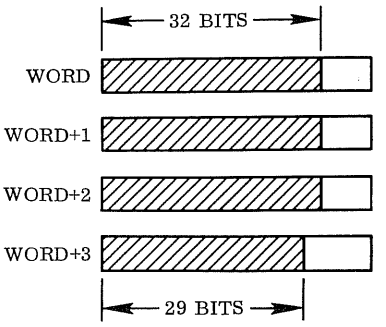
```
TYPE COMPLEX HERCULES
DIMENSION HERCULES (10,20),
```

the number of elements in the array HERCULES is 200. The TYPE statement, however, specifies two words per element; therefore, the number of computer words reserved is 400. The argument is the same for TYPE DOUBLE. For REAL and INTEGER the number of words in an array equals the number of elements in the array.

For subscripted logical variables, up to 32 bits of a computer word are used; each bit represents an element of the logical variable array. The elements are stored left to right in a computer word starting with the most significant bit position. In the statements

```
TYPE LOGICAL XERXES
DIMENSION XERXES (5,5,5),
```

there are 125 elements in the array XERXES and these elements will occupy four sequential words as shown below.



3.2.1 VARIABLE DIMENSIONS

When an array identifier and its dimensions appear as formal parameters in a function or subroutine, the dimensions may be assigned through the actual parameter list accompanying the function reference or subroutine call. The dimensions so assigned must not exceed the maximum array size specified by the DIMENSION statement in the calling program. See section 5.8, Variable Dimensions and Subprograms for details and examples.

3.3 COMMON

Just as an expression may contain sub-expressions, a program may contain, or call upon, subprograms (Chapter V). Such programs must be able to communicate, and they frequently require access to areas of information that they use in common. These areas are specified by the statement COMMON. The general form of this statement is:

COMMON / (B) I_1 / List / (B) I_2 / List . . .

B is a bank designator and has meaning only in the 3600 computer system. It is an unsigned integer constant between 0 and 7, and is ignored in a FORTRAN-63 program executed in the 1604 computer system. When B is omitted in 3600 programs, it is assumed to be the same as B = 0.

I is a COMMON block identifier and it may be up to eight characters in length. It designates either labeled or numbered COMMON blocks and has the form:

$C_1 C_2 \ldots C_p \qquad 1 \leq p \leq 8$

If C_1 is alphabetic the identifier denotes a labeled COMMON block; the remaining characters may be alphabetic or numeric. If all the C_i are numeric, the identifier denotes a numbered COMMON block. If C_1 is numeric, the remaining characters must be numeric.

Examples

Labeled COMMON Identifiers	Numbered COMMON Identifiers
AZ13	1
MAXIMUS	146
Z	3600

List has the form $V_1, V_2 \ldots$, where V_i is of the form identifier (subscript string).

```
COMMON A,B,C†
COMMON/ /A,B,C,D†
COMMON/BLOCK1/A,B/1234/C(10),D(10,10),E(10,10,10)
COMMON/ (1)BLOCKA/D(15),F(3,3),GOSH(2,3,4),Q1
```

3.4 COMMON BLOCKS

The primary purpose of the COMMON block is to provide the programmer with a means of using, in subprograms, certain COMMON areas specified in the main program by referring only to the block desired. Both numbered and labeled blocks may be used for this purpose. Data stored in labeled COMMON blocks by the DATA statement are available to any subprogram using the appropriate labeled block.

3.4.1 COMMON RULES

COM1 COMMON is non-executable and must precede the first executable statement in the program.

COM2 If TYPE, DIMENSION or COMMON appear together, the order is immaterial.

COM3 The identifiers of labeled COMMON blocks are used only for block identification within the compiler; they may be used elsewhere in the program as other kinds of identifiers.

COM4 For any given dimensioned variable, the dimensions may be declared either in a COMMON statement or in a DIMENSION statement. If declared in both, those of the DIMENSION statement override those declared in the COMMON statement.

COM5 At the beginning of program execution, the contents of the COMMON area are undefined unless specified by a DATA statement.

COM6 An identifier in one COMMON block may not appear in another COMMON block. If it does, the identifier is doubly defined.

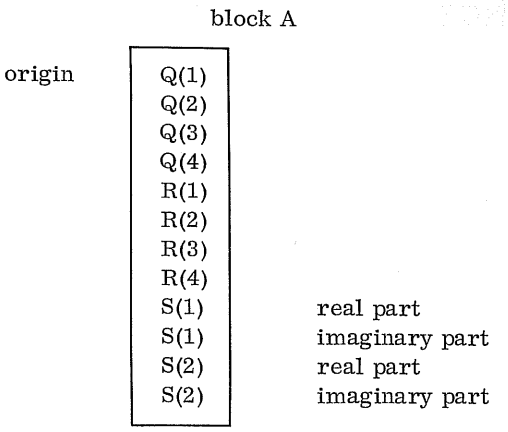
3.4.2 COMMON BLOCK LENGTH

The length of a COMMON block, in computer words, is determined from the type of the list identifier and the dimension (if any) associated with that identifier.

```
Given    COMMON/A/Q(4),R(4),S(2)
         TYPE COMPLEX S
```

the length of the COMMON block A is 12 computer words. The origin of the COMMON block is Q(1).

†These forms are sometimes called blank COMMON.



Examples

```
MAIN PROG     TYPE  COMPLEX C
              COMMON/TEST/C(20)/36/A,B,Z
                .
                .
                .
```

The length of TEST is 40 computer words.

The subprogram may re-arrange the allocation of words as in:

```
SUBPROG1      COMMON/TEST/A(10),G(10),K(10)
              TYPE  COMPLEX A
                .
                .
                .
```

The length of TEST is 40 words. The first 10 elements (20 words) of the block, represented by A, are complex elements. Array G is the next 10 words, and array K is the last 10 words. Within the subprogram, elements of G will be treated as floating point quantities; elements of K will be treated as integer quantities.

The length of the COMMON block must not be changed by the subprograms using the block. The identifiers used within the block may differ as shown above.

The following arrangements are equivalent:

```
{ TYPE DOUBLE A          { TYPE DOUBLE A
{ DIMENSION A(10)        { COMMON A
{ COMMON A               { DIMENSION A(10)

{ DIMENSION A(10)        { TYPE DOUBLE A
{ TYPE DOUBLE A          { COMMON A(10)
{ COMMON A

{ COMMON A
{ DIMENSION A(10)
{ TYPE DOUBLE A
```

The label of a COMMON block is used only for block identification. The following is not erroneous:

```
COMMON /A/A(10)/B/B(5,5) /C/C (5,5,5)
```

3.5 EQUIVALENCE

The EQUIVALENCE statement permits variables to share locations in storage. The general form of this statement is:

EQUIVALENCE (A,B . . .), (A1,B1, . . .), . . .

where the A, B, . . . are simple or singly subscripted variable identifiers. A multiply subscripted variable can be represented by a singly subscripted variable. The correspondence is:

A (i,j,k) ≅ A (the value of (i+(j-1)*I + (k-1)*I*J))

where i,j,k are integer constants and I and J are the integer constants appearing in DIMENSION A (I,J,K). For example, given DIMENSION A(2,3,4), the element A(1,1,2) is represented by A(7).

3.5.1 EQUIVALENCE RULES

EQU1 EQUIVALENCE is non-executable and must precede the first executable statement in the program or subprogram in which it appears.

EQU2 If TYPE, DIMENSION, COMMON, or EQUIVALENCE appear together, the order is immaterial.

EQU3 The following may be made equivalent

COMPLEX / COMPLEX	with or without subscript
COMPLEX / DOUBLE	
COMPLEX or DOUBLE / REAL	
COMPLEX or DOUBLE / INTEGER	
REAL / REAL	
REAL / INTEGER	
DOUBLE / DOUBLE	
INTEGER / INTEGER	
LOGICAL / LOGICAL	unsubscripted only
TYPE 5 / TYPE 5 (non-standard)	
TYPE 6 / TYPE 6 (non-standard)	
TYPE 7 / TYPE 7 (non-standard)	

Any variable of TYPE LOGICAL, 5, 6, or 7 may be made equivalent to one of the standard types, but they must not be subscripted.

EQU4 The EQUIVALENCE statement does not rearrange COMMON, but arrays may be defined as equivalent so that the length of the COMMON block is changed. The origin of the COMMON block must not be changed by the EQUIVALENCE statement.

The following simple cases illustrate changes in block lengths caused by the EQUIVALENCE statement.

Given: Arrays A and B
S_a = subscript of A
S_b = subscript of B

CASE I A, B both in COMMON

a) If A appears before B in the COMMON statement:

Sa ≥ Sb is a permissible subscript arrangement
Sa < Sb is not

b) If B appears before A in the COMMON statement

Sa ≤ Sb is a permissible subscript arrangement
Sa > Sb is not

Block 1

origin ⟶	A (1)		COMMON/1/ A(5), B (7)
	A (2)	B (1)	EQUIVALENCE (A(4), B(3))
	A (3)	B (2)	
	A (4)	B (3)	
	A (5)	B (4)	
		B (5)	
		B (6)	
		B (7)	

Statement EQUIVALENCE (A(3), B(4)) changes the origin of block 1. This is not permitted.

	B(1) ⟵ origin changed	
origin ⟶	A(1)	B(2)
	A(2)	B(3)
	A(3)	B(4)
	A(4)	B(5)

CASE II A in COMMON, B not in COMMON (corresponds to CASE Ia)

Sb ≤ Sa is a permissible subscript arrangement
Sb > Sa is not

Block 1

origin ⟶	A(1)		COMMON /1/A(4)
	A(2)	B(1)	DIMENSION B(5)
	A(3)	B(2)	EQUIVALENCE (A(3), B(2))
	A(4)	B(3)	
		B(4)	
		B(5)	

CASE III B in COMMON, A not in COMMON (corresponds to CASE Ib)

Sa ≤ Sb is a permissible subscript
Sa > Sb is not

Block 1

origin ⟶	B(1)		COMMON/1/ B (4)
	B(2)	A(1)	DIMENSION A (5)
	B(3)	A(2)	EQUIVALENCE (B(2), A(1))
	B(4)	A(3)	
		A(4)	
		A(5)	

CASE IV A, B, not in COMMON

No subscript arrangement restrictions.

3.5.2

GENERAL RULE

Regarding EQUIVALENCE and COMMON - Consider the statement

```
EQUIVALENCE (A(6),B(4),C(3),D(8) )
```

The base of the equivalence is the identifier with the largest subscript. The base is D(8); A(6),B(4), and C(3) will be made equivalent to it.

If any, or all, of A, B, C, D occur in a COMMON statement, the order, from left to right, is by descending subscripts in the EQUIVALENCE statement. Since the subscript of D is greater than the subscript of A, et cetera, the following COMMON statement is permissible:

```
COMMON/1/D(10),A(8),B(5),C(10)
```

The combined statements

```
EQUIVALENCE (A(6),B(4),C(3),D(8) )
COMMON/1/D(10),A(8),B(5),C(10)
```

yield the storage arrangement:

Block 1

origin	D(1)			
	D(2)			
	D(3)	A(1)		
	D(4)	A(2)		
	D(5)	A(3)	B(1)	
	D(6)	A(4)	B(2)	C(1)
	D(7)	A(5)	B(3)	C(2)
equivalence base	D(8)	A(6)	B(4)	C(3)
	D(9)	A(7)	B(5)	C(4)
	D(10)	A(8)		C(5)
				C(6)
				C(7)
				C(8)
				C(9)
				C(10)

Within the EQUIVALENCE statement, the order is immaterial. EQUIVALENCE (A(6),B(4),C(3),D(8))
is the same as EQUIVALENCE(A(6),D(8),C(3),B(4)).

3.6

DATA

The programmer may assign constant values to variables in the program by using the DATA statement either by itself or with a DIMENSION statement. It may be used to store constant values in variables contained in a labeled COMMON block.

The form of the DATA statement is:

$$DATA(I_1 = List),(I_2 = List), \ldots$$

List contains constants only and has the form

$$a_1, a_2, \ldots, K(b_1, b_2, \ldots), c_1, c_2, \ldots$$

where K is an integer constant repetition factor that causes the parenthetical list following it to be repeated K times. I is an identifier representing a simple variable, a variable with integer constant subscripts, an array, or an array with integer variable subscripts.

3.6.1

DATA RULES

DAT1 DATA is non-executable and must precede the first executable statement in any program or subprogram in which it appears.

DAT2 When DATA appears with TYPE, DIMENSION, COMMON or EQUIVALENCE statements, the order is immaterial.

DAT3 DO loop-implying notation is permissible with the restriction that m_3 cannot appear. Short notation may be used for storing constant values in arrays.

DAT4 No array name declared in blank or numbered COMMON or in a variable DIMENSION can belong to a DATA statement.

DAT5 When a signed constant appears in a DATA list, that sign is unary; in DATA (A = -2.), the negative value of the floating point number 2 replaces A. Negative octal constants are prefixed with minus signs. The operator.NOT.may not be used.

DAT6 With identifiers of types real or integer, the corresponding constant in the list must be the same type; in DATA (A = 2). The type of A is not checked, and an integer 2 will replace A.

DAT7 There must be a one-one correspondence between the identifier and the list. This is particularly important in arrays.

```
Consider    COMMON /BLK/ A(3), B
            DATA (A = 1.,2.,3.,4.)
```

The constants 1.,2.,3., are stored in array locations A, A+1, A+2; the constant 4. is stored in location B. If this occurs unintentionally, erroneous results may occur when B is referred to elsewhere in the program.

```
Consider    COMMON / TUP/ C(3)
            DATA (C = 1.,2.)
```

The constants 1., 2. are stored in array locations C and C+1, the contents of C(3), that is, location C+2 are not defined.

DAT8 Use of DATA with a TYPE LOGICAL variable constitutes a special case (the last example below).

Examples

```
DATA (LEDA=15), (CASTOR=16.0), (POLLUX=84.0)

          LEDA        15
                       .
                       .
                       .
        CASTOR        16.0
                       .
                       .
                       .
        POLLUX        84.0

DATA (A(1,3) = 16.239)

    ARRAY A

          A(1,3)      16.239
```

```
DIMENSION B(10)
DATA (B = 77B, -77B, 4(776B, -774B) )
```

ARRAY B
77B
-77B
776B
-774B
776B
-774B
776B
-774B
776B
-774B

```
COMMON /HERA/ C(4)
DATA (C = 3.6, 3(10.5) )
```

ARRAY C
3.6
10.5
10.5
10.5

```
TYPE COMPLEX PROTEUS
DIMENSION PROTEUS (4)
DATA (PROTEUS = 4( (1.0,  2.0) ) )
```

ARRAY PROTEUS
1.0
2.0
1.0
2.0
1.0
2.0
1.0
2.0

```
DIMENSION MESSAGE (3)
DATA (MESSAGE = 3HWHO,  2HIS, 6HSYLVIA)
```

ARRAY MESSAGE
WHO
IS
SYLVIA

This example illustrates how elements of a logical array are stored by the DATA statement.

Given: TYPE LOGICAL L
COMMON / NETWORK / L (4,8)

Store the following matrix of logical elements:

$$L = \begin{vmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{vmatrix}$$

Arrays are stored columnwise.

Elements of logical arrays are stored 32 bits to the word, left to right, left justified with zero fill.

The matrix fits into one computer word as follows:

```
111  110  101  111  011  010  000  100  101  110  100 0...  0
```

and its octal equivalent is

```
7657320456400000
```

Therefore, the appropriate DATA statement is:

```
DATA  (L = 7657320456400000B)
```

CONTROL STATEMENTS 4

Program execution normally proceeds from one statement to the statement immediately following it in the program. Control statements can be used to alter this sequence or cause a number of iterations of a program section.

Control may be transferred to an executable statement only; a transfer to a non-executable statement will result in a program error. During compilation, however, no error will be indicated.

Iteration control provided by the DO statement causes a predetermined sequence of instructions to be repeated any number of times with the stepping of a simple integer variable after each iteration.

4.1 STATEMENT IDENTIFIERS

Statements are identified by numbers which can be referred to from other sections of the program. A statement number used as a label or tag, appears in columns 1 through 5 on the same line as the statement on the coding form. The statement number N may lie in the range $1 \leq N \leq 99999$. An N of fewer than 5 digits may occupy any of the first five columns; blanks are squeezed out and leading zeros are ignored, $1 \cong 01 \cong 001 \cong 0001$, (Appendix A).

4.2 GO TO STATEMENTS

Unconditional transfer of control is provided by GO TO statements.

4.2.1 UNCONDITIONAL GO TO

GO TO n
This statement causes an unconditional transfer to the statement labeled n; n is a statement number.

4.2.2 ASSIGNED GO TO

GO TO m, $(n_1, n_2, \ldots, n_m)$
This statement acts as a many-branch GO TO.
m is an integer variable assigned an integer value n_i in a preceding ASSIGN Statement. The n_i are statement numbers. The parenthetical list need not be present.

4.2.3 ASSIGN STATEMENT

ASSIGN $\mathcal{A}$ TO m
This statement is used with the Assigned GO TO statement.
$\mathcal{A}$ is a statement number, m is a simple integer variable.

```
ASSIGN 10 TO LSWTCH
        .
        .
        .
GO TO LSWTCH,(5,10,15,20)
```

Control would transfer to statement 10.

4.2.4 COMPUTED GO TO

GO TO $(n_1,n_2, \ldots, n_m)$, i
This statement acts as a many-branch GO TO where i is preset or computed prior to its use in the GO TO.
The n_i are statement numbers and i is a simple integer variable. If $i \leq 1$, a transfer to n_1 occurs; if $i \geq m$, a transfer to n_m occurs.

```
    ISWITCH = 1
    GO TO (10,20,30),ISWITCH
          .
          .
          .
10  JSWITCH = ISWITCH + 1
    GO TO (11,21,31),JSWITCH
```

Control would transfer to statement 21.

4.3 IF STATEMENTS

Conditional transfer of control is provided by the two- and three-branch IF statements, the status of sense lights or switches, or the status of an arithmetic overflow indicator.

4.3.1 THREE BRANCH IF (ARITHMETIC)

IF (A) n_1,n_2,n_3
A is an arithmetic expression and the n_i are statement numbers.
This statement tests the evaluated quantity A and jumps according to the following criteria:

A < 0	jump to statement n_1
A = 0	jump to statement n_2
A > 0	jump to statement n_3

In the test for zero, +0 = -0. When the mode of the evaluated expression is complex, only the real part is tested for zero.

```
IF(A*B-C*SINF(X) )10,10,20
IF(I)5,6,7
IF(A/B**2)3,6,6
```

4.3.2 TWO BRANCH IF (LOGICAL)

IF(L) n_1,n_2

L is a logical or an arithmetic expression. The n_i are statement numbers. The evaluated expression is tested for true (non-zero) or false (zero). If L is true jump to statement n_1. If L is false jump to statement n_2.

```
IF(A .GT. 16. .OR. I  .EQ. 0)5,10
IF(L)1,2                          (L is TYPE LOGICAL)
IF(A*B-C)1,2                      (A*B-C is arithmetic)
IF(A*B/C .LE. 14.32)4,6
```

In the statement IF (A) 2,3,4,
A is tested as shown in 4.3.1. In the statement IF (A) 4,3
if A is not zero, jump to statement 4; if A is zero, jump to statement 3.

4.3.3 SENSE LIGHT

IF(SENSE LIGHT i)n_1,n_2

The statement tests sense light i. If it is on, it is turned off and a jump occurs to statement n_1. If it is off, a jump occurs to statement n_2.
i is a sense light and the n_i are statement numbers. i may be a simple integer variable or constant.

```
IF(SENSE LIGHT 4)10,20
```

4.3.4 SENSE SWITCH

IF(SENSE SWITCH i)n_1 ,n_2

If sense switch i is set (ON) a jump occurs to statement n_1 . If it is not set (OFF) a jump occurs to statement n_2 ; i may be a simple integer variable or constant.
In the 3600 $1 \leq i \leq 6$ (physical console switches)
In the 1604 $1 \leq i \leq 6$ (CO OP Monitor function. Appendix E)

```
N = 5
IF(SENSE SWITCH N)5,10
```

4.4 FAULT CONDITION STATEMENTS

At execute time the computer is set to interrupt on divide, overflow or exponent fault.

IF DIVIDE CHECK n_1,n_2
IF DIVIDE FAULT n_1,n_2

A divide fault occurs following division by zero. The statement checks for this fault; if it has occurred, the indicator is turned off and a jump to statement n_1 takes place. If no fault exists, a jump to statement n_2 takes place.

IF EXPONENT FAULT n_1,n_2

An exponent fault occurs when the result of a real or double or complex arithmetic operation exceeds the upper limits specified for these types. Results that are less than the lower limits are set to zero without indication. This statement is therefore a test for floating-point overflow only. If the fault has occurred, the indicator is turned off, and a jump to statement n_1 takes place. If no fault exists a jump to statement n_2 takes place.

IF OVERFLOW FAULT n_1,n_2

An overflow fault occurs when the magnitude of the result of an integer sum or difference exceeds $2^{47}-1$. This fault does not occur in division and it is not indicated in multiplication. If the fault occurs, the indicator is turned off and a jump to statement n_1 takes place. If no fault exists, a jump to statement n_2 takes place.

4.5 DO STATEMENT

DO n i = m_1 ,m_2 ,m_3

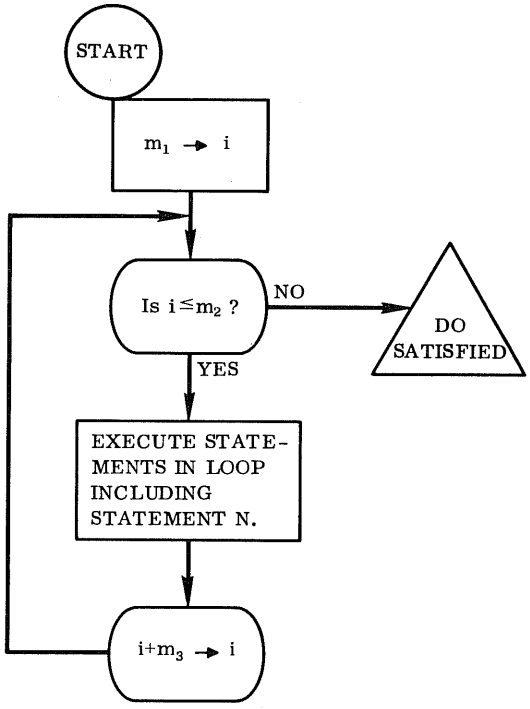
The DO Statement provides FORTRAN-63 with a recursive property.

n is a statement number; i is the index variable. It is a simple integer variable. m_i are the indexing parameters; they may be unsigned integer constants or simple integer variables. If m_3 does not appear, it is construed to be 1.

The DO Statement, the statement labeled n, and any intermediate statements constitute a DO loop. Statement n may not be an IF or GO TO statement or another DO statement. The statement immediately following the DO statement must be executable (Appendix C).

4.5.1 DO INDEX VARIABLE: i

The initial value of i is m_1. This value is compared with m_2 before executing the DO loop and, if it does not exceed m_2, the loop is executed. After this step, i is increased by m_3 and control passes to the top of the loop where i is again compared with m_2; this process continues until i exceeds m_2 as shown below. Control then passes to the statement immediately following n, and the DO loop is said to be satisfied. Should m_1 exceed m_2 on the initial entry to the loop, the loop is not executed and control passes to the next statement.

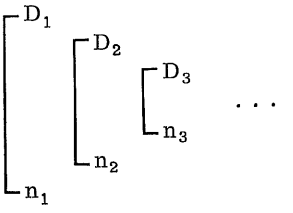


When the DO loop is satisfied, the index variable i is no longer well defined. If a transfer out of the DO loop occurs before the DO is satisfied, the value of i is preserved and may be used in subsequent statements.

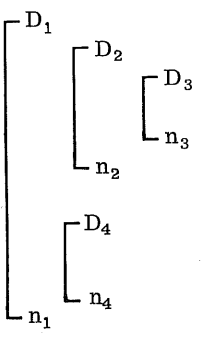
4.5.2

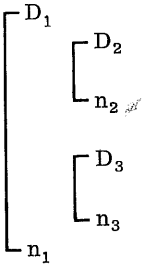
DO NESTS

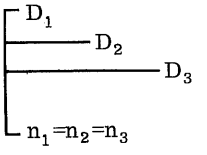
When a DO loop contains another DO loop the grouping is called a DO nest. If $D_1, D_2, \ldots D_m$ represent DO statements, where the subscripts indicate that D_1 appears before D_2 appears before D_3, et cetera, and $n_1, n_2, \ldots, n_m$ represent the corresponding limits of the D_i, then n_m must appear before $n_{m-1} \ldots n_2$ must appear before n_1.



DO loops may be nested in common with other DO loops:







Examples

```
DO 1 I= 1,10,2
   .
   .
   .
DO 2 J=1,5
   .
   .
   .
DO 3 K=2,8
   .
   .
   .
3 CONTINUE
   .
   .
   .
2 CONTINUE
   .
   .
   .
 DO 4 L=1,3
   .
   .
   .
4 CONTINUE
   .
   .
   .
1 CONTINUE
```

```
DO 100 L=2,LIMIT
   .
   .
   .
DO 10 I=1,10
DO 10 J=1,10
   .
   .
   .
10 CONTINUE
   .
   .
   .
 DO 20 K=K1,K2
   .
   .
   .
20 CONTINUE
   .
   .
   .
100 CONTINUE
```

```
DO 5 I=1,5
DO 5 J=I,10
DO 5 K=J,15
   .
   .
   .
5 CONTINUE
```

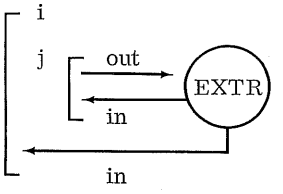
4.5.3
DO LOOP TRANSFER

In a DO nest, a transfer may be made from one DO loop into a DO loop that contains it; and a transfer out of a DO nest is permissible. The special case is transferring out of a nested DO loop and then transferring back to the nest.

In a DO nest:

If the range of i includes the range of j and a transfer out of the range of j occurs, then a transfer into the range of i or j is permissible.

In the following diagram, EXTR represents a portion of the program outside of the DO nest. EXTR must not change the indexing variable or the indexing parameters.



4.5.4
DO PROPERTIES

1) The indexing parameters m_1, m_2, m_3 are either integer constants or simple integer variables.

2) The values of the indexing parameters are assumed to remain constant until the DO is satisfied.

3) The indexing parameters should assume positive values only.

4) If $m_1 > m_2$ initially, the loop is not executed.

5) The identity and value of the indexing variable is local to the statements in the range of the DO statement when

   (a) it is not used as an operand

   (b) No transfers out of the range of the DO exist

   Otherwise, the identity and value of i is global.

6) DO-loops may be nested 50 deep.

4.6
CONTINUE

CONTINUE

The CONTINUE statement is most frequently used as the last statement of a DO loop to provide a transfer address for IF and GO TO instructions that are intended to begin another repetition of the loop. If CONTINUE is used elsewhere in the source program, it acts as a do-nothing instruction and control passes to the next sequential program statement.

4.7 PAUSE

PAUSE

PAUSE n

n is an octal number such that $1 \leq n \leq 2^{47} - 1$. PAUSE n halts the computer with n displayed in the accumulator register on the console. When the START key on the console is pressed, program execution proceeds starting with the statement immediately following PAUSE.

4.8 STOP

STOP

STOP n

n is an octal number such that $1 \leq n \leq 2^{47} - 1$. STOP n halts the computer with n in the accumulator register displayed on the console. When the START key on the console is pressed, an exit will be made to the COOP MONITOR (1604) or SCOPE (3600). STOP (n omitted) causes immediate exit to MONITOR or SCOPE.

4.9 END

END

END marks the physical end of a program or subprogram; if executed, it acts as a RETURN.

FORTRAN-63 functions and subroutines, range from single source language statements to independently compilable subprograms.

A function name is constructed in the same way as a variable identifier and has a type determined by the conventions established for variables. A function together with its arguments may be used at any place in an expression that a variable identifier may be used.

A reference to a function is a call upon a computational procedure for the return of a single value, identified by and associated with the function identifier. This procedure may be defined in a single statement within the program (statement function); it may be defined within the compiler (library function) or it may be defined in a multi-statement subprogram either compiled with a main program or compiled independently (function subroutine).

A reference to a subroutine is also a call upon a computation procedure. This procedure may return one or more values or it may return none. No value is associated with the name of the subroutine, and the subroutine must be called by a CALL statement.

Any function reference must supply the function with a set of arguments or parameters. This set must contain at least one argument and may contain up to 63 arguments. The forms of the arguments differ somewhat in each of the three kinds of functions. The form of the function reference is:

$$F\ (p_1,p_2,\ldots p_n) \qquad 1 \leq n \leq 63$$

where F is the function name and the p_i are function arguments or <u>actual</u> parameters. The corresponding arguments appearing with the function name in a function definition are called <u>formal</u> parameters.

5.1 STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic or logical statement in the source program and apply only to the particular program or subprogram in which the definition appears. They have the form

$$F\ (p_1,p_2,\ldots p_n) = E$$

where F is the function name and E is an expression.

5.1.1 STATEMENT FUNCTION RULES

SF1 The type of the function is determined from the naming conventions specified for variables in Chapter 3, Type Declarations.

SF2 The function name must not appear in a DIMENSION, EQUIVALENCE or COMMON statement.

SF3 The formal parameters will usually appear in the expression E. When the statement function is executed, the formal parameters are replaced by the corresponding actual parameters of the function reference. Each of the formal parameters may be TYPE REAL or INTEGER only, but they may not be declared in a TYPE statement. (3.1.1, rule TD3) Each of the actual parameters may be any arithmetic expression, but there must be agreement in order, number and type between the actual and formal parameters.

SF4 E may be arithmetic or logical.

SF5 E cannot contain subscripted variables.

SF6 The expression E may refer to library functions, previously defined statement functions and function subprograms.

SF7 All statement functions must precede the first executable statement of the program or subprogram, but they must follow all declarative statements (DIMENSION, TYPE, et cetera).

Examples

```
TYPE COMPLEX Z
Z(X,Y)=(1.,0.)*EXPF(X)*COSF(Y)+(0.,1.)*EXPF(X)*SINF(Y)
```

This arithmetic statement function computes the complex exponential $Z(x,y) = e^{x+iy}$.

5.2 LIBRARY FUNCTIONS

FORTRAN-63 contains the standard library functions available in earlier versions of FORTRAN. A list of these functions is in Appendix D. The identifying names have not been changed. When one appears in the source program, a special part of the compiler identifies it as a library function and takes appropriate action as explained below.

In Chapter 3, specific rules are given for declaring the types of identifiers. In the absence of a TYPE declaration, a variable type is determined by its first identifier letter. As stated in rule SF1, this convention applies to function identifiers. In the standard library function for obtaining the natural logarithm of a number (LOGF) the first identifier letter, L, would cause that function to return an integer result. In this case the result is contrary to established FORTRAN usage. To avoid inconsistency, the compiler recognizes the standard library functions and permits the programmer to use such functions in the usual manner.

5.3 FUNCTION SUBPROGRAMS

Function subprograms are FORTRAN source language programs that cannot be defined by one statement and that are not used frequently enough to be included in the library.

Function subprograms may be compiled independently, and the first statement of such a subprogram must have the form:

FUNCTION F $(p_1,p_2,\ldots p_n)$ $1 \leq n \leq 63$

where F is the function name, and the p_i are formal parameters. These parameters may be array names, non-subscripted variables, or names of other function or subroutine subprograms.

5.3.1 FUNCTION SUBPROGRAM RULES

FS1 The type of the function is determined from the naming conventions specified for variables in Chapter 3, Type Declarations.

FS2 The name of a function must not appear in a DIMENSION statement. The name must appear, however, at least once as any of the following:

The left-hand identifier of a replacement statement

An element of an input list

An actual parameter of a subprogram call

FS3 No element of a formal parameter list may appear in a COMMON or EQUIVALENCE statement within the function subprogram.

FS3A When a formal parameter represents an array, it should be declared in a DIMENSION statement within the function subprogram. If it is not declared, only the first element of the array will be available to the function subprogram.

FS4 In referring to a function subprogram the following forms of the actual parameters are permissible:

a. arithmetic expression

b. constant or variable, simple or subscripted

c. array name

d. function reference

e. subroutine

In form d, if only the name of the function appears, it must also appear in an EXTERNAL statement in the calling program. See example 2 following 5.5.

In form e, the subroutine may appear as a subroutine name alone or as a subroutine name with a parameter bit:

These cases are illustrated in examples 3 and 4 following 5.5.

FS4A Logical expressions may not be actual parameters.

FS5 Actual and formal parameters must agree in order, number and type.

5.3.2 FUNCTION TYPE AND MODE

The compiler distinguishes between array names and functions as follows: Given an identifier followed by a left parenthesis, Z(, if the identifier Z occurs in a DIMENSION statement, it represents an array. If not, Z(represents a function. To determine the mode of the function (statement, subprogram, or library), the compiler goes through the following process of elimination:

1. If the identifier is not in the compiler's table of library functions, or is not declared in a TYPE statement, the mode of the evaluated function is according to the identifier first-letter criterion (3.1.1).

2. If the identifier is in the compiler's table of library functions, but not in a TYPE statement, the mode of the evaluated function is given by the library function (Appendix D).

3. If the name is declared in a TYPE statement, the mode of the evaluated function is defined by the TYPE statement (3.1.1).

5.4 RETURN AND END STATEMENTS

A subprogram normally contains one or several RETURN statements that indicate the end of logic flow within the subprogram, and return control to the calling program. In function references, control returns to the statement in which the function is imbedded. In subroutine subprograms, control returns to the next executable statement immediately following the CALL statement in the calling program. The form of this statement is RETURN.

The END statement marks the physical end of a program, subroutine subprogram or function subprogram. If the RETURN statement is omitted, END acts as a return to the calling program.

5.5 EXTERNAL STATEMENT

When the actual parameter list of a given function reference contains a function or subroutine name, that name must be declared in an EXTERNAL statement. Its form is:

EXTERNAL identifier$_1$, identifier$_2$, . . .

where identifier is the name of a function or subroutine. The EXTERNAL statement must precede the first executable statement of any program in which it appears. When it is used, EXTERNAL always appears in the calling program.

Examples

1. Function Subprogram

```
      FUNCTION GREATER (A,B)
      IF (A .GT. B) 1,2
    1 GREATER = A-B
      RETURN
    2 GREATER = A+B
      END
```

Calling Program Reference

```
      Z(I,J) = F1+F2-GREATER (C-D,3.*I/J)
```

2. Function Subprogram

```
      FUNCTION PHI (ALFA, PHI2)
      PHI = PHI2(ALFA)
      END
```

Calling Program Reference

```
EXTERNAL SINF
   .
   .
   .
C=D-PHI (Q(K),SINF)
```

From its call in the main program, the formal parameter ALFA is replaced by Q(K), and the formal parameter PHI2 is replaced by SINF. PHI will be replaced by the sine of Q(K).

3. Function Subprogram

```
FUNCTION PSYCHE (A,B,X)
CALL X
PSYCHE = A/B*2*(A-B)
END
```

Function Subprogram Reference

```
EXTERNAL EROS
        .
        .
        .
R = S - PSYCHE (TLIM,ULIM,EROS)
```

In the function subprogram, TLIM, ULIM replaces A,B. The CALL X is a call to a subroutine named EROS. EROS appears in an EXTERNAL statement so that the compiler recognizes it as a subroutine name rather than a variable identifier.

4. Function Subprogram

```
FUNCTION AL(W,X,Y,Z)
CALL W(X,Y,Z)
AL = Z**4
RETURN
```

Function Subprogram Reference

```
EXTERNAL SUM
        .
        .
        .
G = AL(SUM,E,V,H)
```

In the function subprogram the name of the subroutine (SUM) and its parameters (E,V,H) replace W and X,Y,Z. SUM appears in the EXTERNAL statement so that the compiler will treat it as a subroutine name rather than a variable identifier.

5.6 SUBROUTINE SUBPROGRAMS

Subroutine subprograms may be compiled independently; the first statement of such a program must have the form:

SUBROUTINE S

or

SUBROUTINE S $(p_1,p_2, \ldots p_n)$ $1 \leq n \leq 63$

where S is the subroutine name, and the p_i are the formal parameters which may be array names, non-subscripted variables, or names of other function or subroutine subprograms.

5.6.1 SUBROUTINE RULES

SS1 The name of the subroutine may not appear in any declarative statement (TYPE, DIMENSION) in the subroutine.

SS2 The name of the subroutine must never appear within the subroutine as an identifier in a replacement statement, in an input-output list, or as an argument of another CALL.

SS3 No element of a formal parameter list may appear in a COMMON or EQUIVALENCE statement within the subroutine subprogram.

SS3A When a formal parameter represents an array, it should be declared in a DIMENSION statement within the subroutine. If it is not declared, only the first element of the array will be available to the subroutine.

The executable statement in the calling program for referring to a subroutine subprogram is of the form:

CALL S

or

CALL S $(p_1,p_2, \ldots p_n)$ $1 \leq n \leq 63$

where S is the subprogram name, and the p_i are the actual parameters.

5.6.2 SUBROUTINE REFERENCE RULES

SS4 The subroutine returns values through parameters or COMMON variables. No value is associated with its name.

SS5 The subroutine name may not appear in any declarative statement (TYPE, DIMENSION et cetera).

SS6 In the subroutine call, the following forms of actual parameters are permissible:

a. arithmetic expression

b. constant or variable, simple or subscripted

c. array name

d. function reference

e. subroutine

In form d, if only the name of the function appears, it must also appear in an EXTERNAL statement in the calling program (example 2, 5.5). In form e. the subroutine may appear as a subroutine name alone or as a subroutine name with a parameter list.

SS6A Logical expressions may not be actual parameters.

SS7 Actual and formal parameters must agree in order, number and type.

Examples

1. Subroutine Subprogram

```
      SUBROUTINE BLVDLDR (A,B,W)
      W = 2. *B/A
      END
```

Calling Program References

```
      CALL BLVDLDR (X(I),Y(I),W)
          .
          .
          .
      CALL BLVDLDR (X(I)+H/2.,Y(I)+C(1)/2.,W)
          .
          .
          .
      CALL BLVDLDR (X(I)+H,Y(I)+C(3),Z)
```

2. Subroutine Subprogram (Matrix Multiply)

```
      SUBROUTINE MATMULT
      COMMON/BLK1/X(20,20),Y(20,20),Z(20,20)
      DO   10   I=1,20
      DO   10   J=1,20
      Z(I,J) = 0.
      DO   10   K=1,20
10    Z(I,J) = Z(I,J) + X(I,K) *Y (K,J)
      RETURN
      END
```

Calling Program Reference

```
      COMMON/BLK1/A(20,20),B(20,20),C(20,20)
          .
          .
          .
      CALL MATMULT
          .
          .
          .
```

3. Subroutine Subprogram

```
SUBROUTINE ISHTAR   (Y,Z)
COMMON/1/X(100)
Z = 0.
DO 5 I=1,100
5 Z = Z+X(I)
CALL Y
RETURN
END
```

Calling Program Reference

```
COMMON/1/A(100)
EXTERNAL PRNTIT
   .
   .
   .
CALL ISHTAR (PRNTIT,SUM)
```

5.7 MAIN PROGRAM AND SUBPROGRAMS

A program may be written without references to subprograms or functions. On the other hand, it may refer to subroutines or functions or both. If so, the program is known as the main program. In either instance the first statement must be of the form:

```
PROGRAM name
```

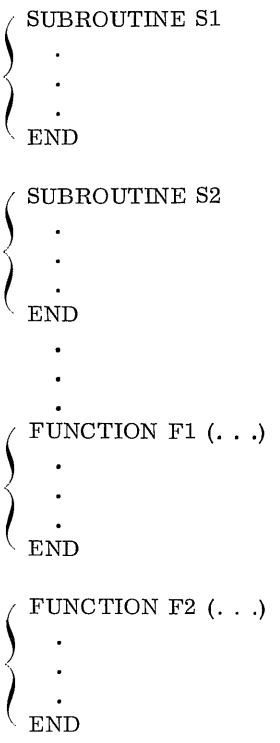
where name is an alphanumeric identifier.

A main program may refer to a variety of subroutines and functions. The subprograms so referred to may be compiled with the main program or independently of the main program. Subprograms compiled with the program are called internal subprograms and give rise to the new terms global and local.

An internal subprogram may be compiled with a main program, a subroutine subprogram, or a function subprogram. The first statement in the respective cases must be PROGRAM, SUBROUTINE S or SUBROUTINE S (p_1 , . . .p_n), or FUNCTION F (p_1 , . . . ,p_n), where the elements of the statement have established definitions. The method of reference to function or subroutine subprograms is the same as stated earlier in this chapter.

FORTRAN-63 assumes that all statements appearing between a PROGRAM, SUBROUTINE or FUNCTION statement and an END statement belong to one program. A typical arrangement of a set of programs and subprograms follows.

```
PROGRAM SOMTHING
   .
   .
   .
END
```



Identifiers that are available to the entire set of programs are called global identifiers. Identifiers that are available only to a particular internal subprogram are called local identifiers.

5.7.1 ENTRY STATEMENT

This statement provides alternate entry points to a function or subroutine subprogram. Its form is

ENTRY name

where name is an alphanumeric identifier, and may appear within the subprogram only in the ENTRY statement. Up to 19 entries are permitted to a given subprogram by use of this statement, but each entry identifier must appear in a separate ENTRY statement. The formal parameters, if any, appearing with the FUNCTION or SUBROUTINE statement do not appear with the ENTRY statement. The ENTRY statement may appear anywhere within the subprogram.

In the calling program, the reference to the entry name is made just as if reference was being made to the FUNCTION or SUBROUTINE in which the ENTRY is imbedded. Rules FS4 and FS4A of 5.3.1 apply.

Examples

```
      FUNCTION JOE(X,Y)
10    JOE = X+Y
      RETURN
      ENTRY SAM
      IF  (X .GR. Y) 10,20
20    JOE = X-Y
      END
```

This could be called from the main program as follows:

```
.
.
.
Z = A+B-JOE  (3.*P,Q-1.)
.
.
.
R=S+SAM(Q,2.*P)
.
.
.
```

5.7.2 NON-RECURSIVENESS OF SUBPROGRAMS

Subprograms may be called from a main program or from other subprograms. Any subprogram called, however, may not call the calling program. That is, if program A calls subprogram B, subprogram B may not call program A. Furthermore, a program or subprogram may not call itself.

5.8 VARIABLE DIMENSIONS AND SUBPROGRAMS

In many subprograms, especially those performing matrix manipulation, the programmer may wish to vary the dimension of the arrays each time the subprogram is called. This is accomplished by specifying the array identifier and its dimensions as formal parameters in the function or subroutine statement heading a function or subroutine subprogram. In the subroutine call from the calling program, the corresponding actual parameters are specified, and these values are used by the called subprogram.

5.8.1 VARIABLE DIMENSION RULES

VAR1 The formal parameters representing the array dimensions must be simple integer variables in a DIMENSION statement within the subprogram. The array identifier must also be a formal parameter.

VAR2 The actual parameters representing the array dimensions may be integer constants, integer variables, or integer arithmetic expressions.

VAR3 If the total number of elements of a given array in the calling program is N, then the total number of elements of the corresponding array in the subprogram should not exceed N.

Example

1. Consider a simple matrix add routine written as a subroutine subprogram:

```
      SUBROUTINE MATADD (X,Y,Z,M,N)
      DIMENSION X (M,N), Y(M,N), Z(M,N)
      DO  10  I=1,M
      DO  10  J=1,N
10    Z(I,J) = X(I,J) + Y(I,J)
      RETURN
      END
```

The arrays X, Y, Z and the variable dimensions M, N must all appear as formal parameters in the SUBROUTINE statement and also appear in the DIMENSION statement as shown. If the calling program contains the array allocation declaration:

```
DIMENSION A(10,10), B(10,10), C(10,10)
```

the program may call the subroutine MATADD from several places within the main program, varying the array dimension within MATADD each time as follows.

```
CALL MATADD  (A, B, C, 3, 6)
             .
             .
             .
CALL MATADD  (A, B, C, 4, 8)
             .
             .
             .
CALL MATADD  (A, B, C, 3, 12)
             .
             .
             .
CALL MATADD  (A, B, C, 4*LIM, LIM2+3)
```

When the actual parameters representing the array dimensions are integer expressions, the limits of the array established by the DIMENSION statement in the main program may be exceeded. This condition is not checked by the compiler.

2. Consider the 4 by n matrix:

$$Y = \begin{matrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2n} \\ y_{31} & \cdots & y_{3n} \\ y_{41} & \cdots & y_{4n} \end{matrix}$$

Its transpose Y^1 is:

$$Y^1 = \begin{matrix} y_{11} & y_{21} & y_{31} & y_{41} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ y_{1n} & y_{2n} & y_{3n} & y_{4n} \end{matrix}$$

The following FORTRAN-63 program permits variation of n from call to call:

```
    SUBROUTINE MATRAN (Y, YPRIME, N)
    DIMENSION Y (4, N), YPRIME (N, 4)
    DO 7 I=1, 4
    DO 7 J=1, N
  7 YPRIME (I, J) = Y (J, I)
    END
```

APPENDIX SECTION

APPENDIX A

CODING PROCEDURES

CODING FORM

FORTRAN-63 forms contain 80 columns in which the characters of the language are written, one character per column.

STATEMENTS

The statements of FORTRAN-63 are written in columns 7 through 72. Statements longer than 66 columns may be carried to the next line by using a continuation designator. Statements may be compacted, several to a given line. Blanks may be used freely in FORTRAN statements to provide readability. Blanks are significant, however, in H fields.

STATEMENT SEPARATOR $

The special character $ may be used to write more than one statement on a line. Statements so written may also use the CONTINUATION feature.

These statements are equivalent:

```
I = 10                          I = 10 $ JLIM = 1 $ K = K+1 $ GO TO 10
JLIM = 1
K = K+1
GO TO 10
```

Also:

```
DO 1 I=1, 10                    DO 1 I=1, 10 $ A(I)=B(I)+C(I)
A(I)=B(I)+C(I)                1 CONTINUE $ I=3
1 CONTINUE
I=3
```

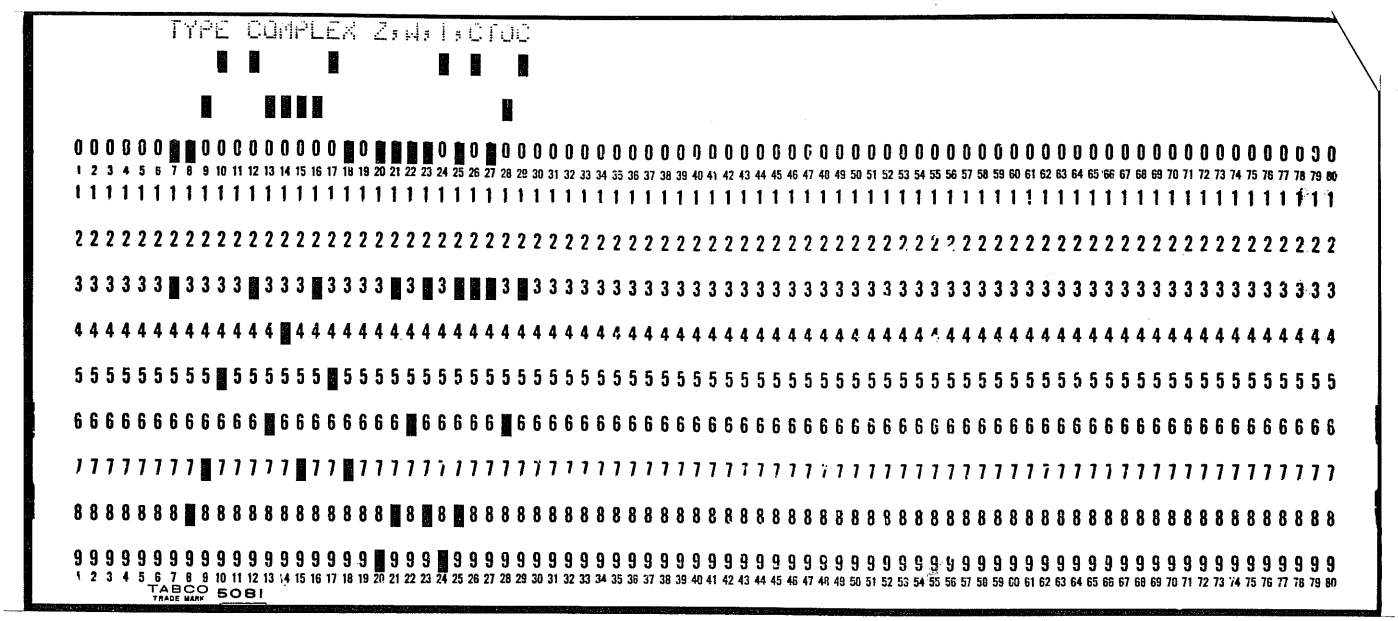
COMMENT CARD

Comment information is designated by a C in column 1 of a statement. Comment information will appear in the source program, but it is not translated into object code. Columns 2 through 80 may be used. Continuation is not permitted; that is, each line of comments must be preceded by the column 1 C designator.

All comment cards belonging to a specific program, or subprogram, should appear between the PROGRAM, SUBROUTINE, or FUNCTION statement and the END statement.

STATEMENT IDENTIFIERS

Any statement may have an identifier (tag, label, number) but only statements referred to elsewhere in the program require identifiers. A statement identifier is a string of from 1 to 5 digits occupying any column positions 1 through 5.



Hollerith Card

1604 FORTRAN CODING FORM

PROGRAM

ROUTINE

NAME

PAGE

DATE

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT	SERIAL NUMBER
			0 = ZERO I = ONE 2 = TWO Ø = ALPHA O I = ALPHA I Z = ALPHA Z	
			FUNCTION CTØC(Z,W)	
			TYPE COMPLEX Z,W,I,CTØC	
			TYPE REAL DATA (PI=1.5707963268) ,(I=(0.,1.))	
			A=Z $ B=-I*Z $ C=W $ D=-I*Z	
			IF(A) 20,,10	
10			P=B $ THETA=PI $ GØ TØ 30	
20			THETA=ATAN(B/A)	
			R=SQRTF(A*A+B*B)	
30			LØGR=LØGR(R)	
			CTØC=EXPF(C*LØGR-D*THETA)*(COSF(D*LØGR+C*THETA)+	
	*		SINF(D*LØGR+C*THETA)*I)	
			RETURN	
			END	

FORM 252-A

FORTRAN Coding Form

If I is such an identifier, $1 \leq I \leq 99999$; lead zeros are ignored, $1 \cong 01 \cong 001 \cong 0001$. Zero is not a statement identifier. In any given program or subprogram each statement identifier must be unique. If the statement identifier is followed by a character other than zero in column 6 the statement identifier is ignored.

CONTINUATION

The first line of every statement must have a blank in column 6. If statements occupy more than one line, all subsequent lines must have a FORTRAN character other than blank or zero in column 6. A FORTRAN-63 statement may have up to 600 alphanumeric characters, operators, delimiters (commas or parentheses) and identifiers within it; blanks are not included in this count. In general, up to 20 continuations may appear after a first statement.

IDENTIFICATION FIELD

Columns 73 through 80 are always ignored in the translation process. These columns, therefore, may be used for identification when the program is to be punched on cards. Usually these columns contain sequencing information provided by the programmer.

PUNCHED CARDS

Each line of the coding form corresponds to one 80-column card, and the terms line and card are often used interchangeably. Source programs and data can be read into the computer from cards; an object program memory map, or data, can be punched directly onto cards. Usually, however, cards are used in the off-line preparation of input or output magnetic tapes.

Blank cards appearing within the input card deck are treated as follows:

a) If a blank card appears between a statement and its continuation, the continuation and other continuations following it are lost. Compilation continues.

b) If a blank card appears between two statements, it is ignored.

When cards are being used as a data medium rather than as statements, all 80 columns may be used.

MAGNETIC TAPE

Magnetic tapes are the most commonly used input-output media. Tape characteristics are described in the Control Data Corporation hardware manuals. The record structure resulting from the READ/WRITE or BUFFER I/O control statements is described in Chapter 2 of Volume II.

Compilation of FORTRAN-63 programs requires the following tapes:

Master tape
Standard input unit
Standard output unit
FORTRAN scratch tape } not used for very
Assembler scratch tape } small programs
Standard punch unit

CARRIAGE CONTROL

The first character of the printer record is a control character for providing line spacing control. This character is not printed but will cause the following actions:

Character	Action
Blank	Single space before printing
0	Double space before printing
1	Eject page before printing

These codes are standard on all printers used with the 1604. Some printers provide additional codes which are given in the specific manuals.

APPENDIX B

CHARACTER CODES 1604 COMPUTER

Source Language Character	BCD (Magnetic Tape & Internal)	Punch Positions in a Hollerith Card Column
A	61	12-1
B	62	12-2
C	63	12-3
D	64	12-4
E	65	12-5
F	66	12-6
G	67	12-7
H	70	12-8
I	71	12-9
J	41	11-1
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
S	22	0-2
T	23	0-3
U	24	0-4
V	25	0-5
W	26	0-6
X	27	0-7
Y	30	0-8
Z	31	0-9
0	12	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
/	21	0-1
+	60	12
-	40,14	11,8-4
blank	20	space
.	73	12-8-3
)	74	12-8-4
$	53	11-8-3
*	54	11-8-4
,	33	0-8-3
(	34	0-8-4
=	13	8-3

CHARACTER CODES 3600 COMPUTER

Source Language Character	BCD (Internal only)*	Punch position in a Hollerith Card Column
A	21	12-1
B	22	12-2
C	23	12-3
D	24	12-4
E	25	12-5
F	26	12-6
G	27	12-7
H	30	12-8
I	31	12-9
J	41	11-1
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
S	62	0-2
T	63	0-3
U	64	0-4
V	65	0-5
W	66	0-6
X	67	0-7
Y	70	0-8
Z	71	0-9
0	00	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
/	61	0-1
+	20	12
-	40	11-8-4
blank	60	space
.	33	12-8-3
)	34	12-8-4
$	53	11-8-3
*	54	11-8-4
,	73	0-8-3
(	74	0-8-4
=	13	8-3

*Magnetic Tape Codes same as 1604.

APPENDIX C

STATEMENTS OF FORTRAN-63

			Page	
REPLACEMENT	A=E Arithmetic	E*	14	Volume I
	A=L Logical	E	20	
	A=M Masking	E	21	
	$A_m=\ldots=A_1$=E Multiple	E	22	
CONTROL	GO TO n	E	37	
	GO TO n,($n_1, \ldots n_m$)	E	37	
	GO TO ($n_1, \ldots, n_m$),i	E	38	
	ASSIGN i to n	E	37	
	IF(A)n_1, n_2, n_3	E	38	
	IF(ℓ)n_1, n_2	E	39	
	SENSE LIGHT i	E	39	
	IF(SENSE LIGHT i)n_1, n_2	E	39	
	IF(SENSE SWITCH i)n_1, n_2	E	39	
	IF DIVIDE $\left\{\begin{matrix}\text{FAULT}\\ \text{CHECK}\end{matrix}\right\}$ n_1, n_2	E	39	
	IF EXPONENT FAULT n_1, n_2	E	39	
	IF OVERFLOW FAULT n_1, n_2	E	40	
	DO n i=m_1, m_2, m_3	E	40	
	CONTINUE	E	42	
	PAUSE	E	43	
	STOP	E	43	
	END	N/E	43	
TYPE DECLARATION	TYPE COMPLEX List	N	25	
	TYPE DOUBLE List	N	25	
	TYPE REAL List	N	25	
	TYPE INTEGER List	N	25	
	TYPE LOGICAL List	N	25	
	TYPE named (W/f) List	N	25	
	d is 5, 6, or 7			

* E = Executable N = Non-executable

			Page	
STORAGE ALLOCATION	DIMENSION $V_1, V_2, \ldots$	N	32	Volume I
	COMMON/B(I)/List ...	N	27	
	EQUIVALENCE(a,b,c. . .)(p,q. . .)	N	30	
DATA STATEMENT	DATA(I=List),(I=List), ...	N	32	
SUBPROGRAM STATEMENTS	FUNCTION name($p_1, p_2, \ldots$)	N	45	
	SUBROUTINE name($p_1, p_2, \ldots$)	N	46	
	PROGRAM name	N	47	
	EXTERNAL $name_1, name_2, \ldots$	N	48	
	ENTRY name	N	53	
	CALL name	E	48	
	RETURN	E	48	
I/O FORMAT	FORMAT($spec_1, spec_2, \ldots$)	N	Volume II	
I/O READ/WRITE	READ n,I	E	25	
	PRINT n,L	E	21	
	PUNCH n,L	E	21	
	READ(i,n)L READ INPUT TAPE i,n,L	E	21	
	WRITE(i,n)L WRITE OUTPUT TAPE i,n,L	E	21	
	READ(i)L READ TAPE i,L	E	25	
	WRITE(i)L WRITE TAPE i,L	E	25	
I/O TAPE HANDLING	END FILE i	E	26	
	REWIND i	E	26	
	BACKSPACE i	E	26	
I/O STATUS CHECKING	IF (EOF,i)n_1, n_2	E	28	
	IF (IOCHECK,i)n_1, n_2	E	28	
	IF (UNIT,i)n_1, n_2, n_3, n_4	E	28	
I/O BUFFERING	BUFFER IN(i,p)(A,B) BUFFER OUT (i,p)(A,B)	E	26	
INTERNAL DATA MANIPUATION	ENCODE (c,n,V)L	E	29	
	DECODE (c,n,V)L	E	29	

APPENDIX D

LIBRARY FUNCTIONS

Form	Definition	Actual Parameter type	Mode of Result
ABSF(X)	Absolute Value	Real	Real
XABSF(i)INTF(X)		Integer	Integer
INTF(X)	Truncation, integer	Real	Real
XINTF(X)		Real	Integer
MODF(X_1,X_2)	X_1 modulo X_2	Real	Integer
XMODF(i_1,i_2)	i_1 modulo i_2	Integer	Integer
MAX0F(i_1,i_2 . . .)	Determine maximum argument	Integer	Real
MAX1F(X_1,X_2, . . .)		Real	Real
XMAX0F(i_1,i_2, . . .)		Integer	Integer
SMAX1F(X_1,X_2, . . .)		Real	Integer
MIN0F(i_1,i_2, . . .)	Determine minimum argument	Integer	Real
MIN1F(X_1,X_2, . . .)		Real	Real
XMIN0F(i_1,i_2, . . .)		Integer	Integer
XMIN1F(X_1,X_2, . . .)		Real	Integer
SINF(X)	Sine X radians	Real	Real
COSF(X)	Cosine X radians	Real	Real
TANF(X)	Tangent X radians	Real	Real
ASINF(X)	Arcsine X radians	Real	Real
ACOSF(X)	Arccos X radians	Real	Real
ATANF(X)	Arctangent X radians	Real	Real
TANHF(X)	Hyperbolic tangent X radians	Real	Real
SQRTF(X)	Square root of X	Real	Real
LOGF(X)	Natural log of X	Real	Real
EXPF(X)	e to xth power	Real	Real
SIGNF(X_1,X_2)	Sign of X_2 times X_1	Real	Real
XSIGNF(i_1,i_2)	Sign of i_2 times i_1	Integer	Integer
DIMF(X_1,X_2)	If $X_1 > X_2$: $X_1 - X_2$ If $X1 \leq X_2$: 0	Real	Real
XDIMF(i_1,i_2)	If $i_1 > i_2$: $i_1 - i_2$ If $i_1 \leq i_2$: 0	Integer	Integer

Form	Definition	Actual Parameter Type	Mode of Result
CUBERTF (X)	Cube root of X	Real	Real
FLOATF(I)	Integer of Real Conversion	Integer	Real
RANF(N)	Generate Random Number	-Real -Integer	Real
	(Repeated Executions give uniformly distributed numbers)	+Real +Integer	Integer
XFIXF	Real to Integer Conversion	Real	Integer
POWERF(X_1,X_2)	$X_1^{X_2}$	Real, Real	Real
ITOJ(I,J)	I^J	Integer, Integer	Integer
XTOI(X,I)	X^I	Real, Integer	Real
ITOX(I,X)	I^X	Integer, Real	Real

INDEX - Volume I

Errata Sheet

FORTRAN-63/Reference Manual - Volume I continued - Publication Number 527

Page	Article	Remarks
59	Appendix A	FORTRAN Coding Form, Statement 30 should read "LOGR = LOGF(R)"
63	Appendix B	Punch position for minus is 11,8-4
66	Appendix D	SMAX1F should be XMAX1F

Errata Sheet

FORTRAN-63/Reference Manual Volume 1 Publication #527

Page	Article	
17	2.5	First line: strike the character "/".
27	3.3	Last paragraph before examples. Add the sentence "Leading zeros in numeric identifiers are ignored."
32	3.5.2	Second paragraph; change "If any, or all," to "If all".
33	3.6.1	Rule DAT6, 2nd line should read in part: "...DATA (A=2), the type ..."
39	4.3.3	The F63 statement SENSE LIGHT is omitted. It reads: "SENSE LIGHT i The statement turns on the i-th sense light. SENSE LIGHT 0 turns off all sense lights. i may be a simple integer variable or constant. In the 3600, $1 \leq i \leq 48$ In the 1604, $1 \leq i \leq 4$."
39	4.3.4	Strike the words "Appendix E".
40	4.5	Strike out the last sentence of the last paragraph. Strike out the sentence "The DO statement provides FORTRAN-63, etc."
46	5.1	Rule SF3. Add: "Formal parameters must be simple variables" Rule SF5. Change to: "E may contain subscripted variables, but the subscripts are restricted to integer constants."
47	5.3.1	Rule FS4, next to last sentence--last word should be "list", not "bit".
49	5.5	Example 4: "END" follows "RETURN"
53	5.7.1	In the Example: Change "ENTRY SAM" to "ENTRY JAM" Change "IF (X.GR.Y)10,20" to "IF(X.GT.Y)10,20".
54	5.7.1	Change "S + SAM(Q,2.*P)" to "S + JAM(Q,2.*P)"
54	5.8.1	Add to rule VAR1: "The formal parameters must not appear in a COMMON or EQUIVALENCE statement in the subprogram".
55	5.8.1	Example 2 Change "DO 7 I=1,4 to "DO 7 I=1,N DO 7 J=1,N" DO 7 J=1,4"

CONTROL DATA SALES OFFICES

ALBUQUERQUE • BEVERLY HILLS • BIRMINGHAM • BOSTON
CHICAGO • CLEVELAND • DALLAS • DAYTON
DENVER • DETROIT • HONOLULU • HOUSTON
HUNTSVILLE • ITHACA • KANSAS CITY, KAN. • LOS ALTOS • MINNEAPOLIS • NEWARK
NEW YORK CITY • OMAHA • ORLANDO • PALO ALTO • PHILADELPHIA • PITTSBURGH
SAN DIEGO • SAN FRANCISCO • SEATTLE • WASHINGTON, D.C.

INTERNATIONAL OFFICES

BAD HOMBURG, GERMANY • MELBOURNE, AUSTRALIA • LUCERNE, SWITZERLAND
STOCKHOLM, SWEDEN • ZURICH, SWITZERLAND • PARIS, FRANCE • OSLO, NORWAY

CONTROL DATA
CORPORATION

8100 34th AVENUE SOUTH, MINNEAPOLIS 20, MINNESOTA

Pub. No. 527