CONTROL DATA

1604/1604-A COMPUTER

1604/1604-A

FORTRAN 63/REFERENCE MANUAL

CONTROL DATA 1604/1604-A COMPUTER

FORTRAN 63/REFERENCE MANUAL

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

Any comments concerning this manual should be addressed to:

CONTROL DATA CORPORATION
Documentation and Evaluation Department
3145 Porter Drive
Palo Alto, California

June, 1964
Pub. No. 60052900
Revision A

PREFACE

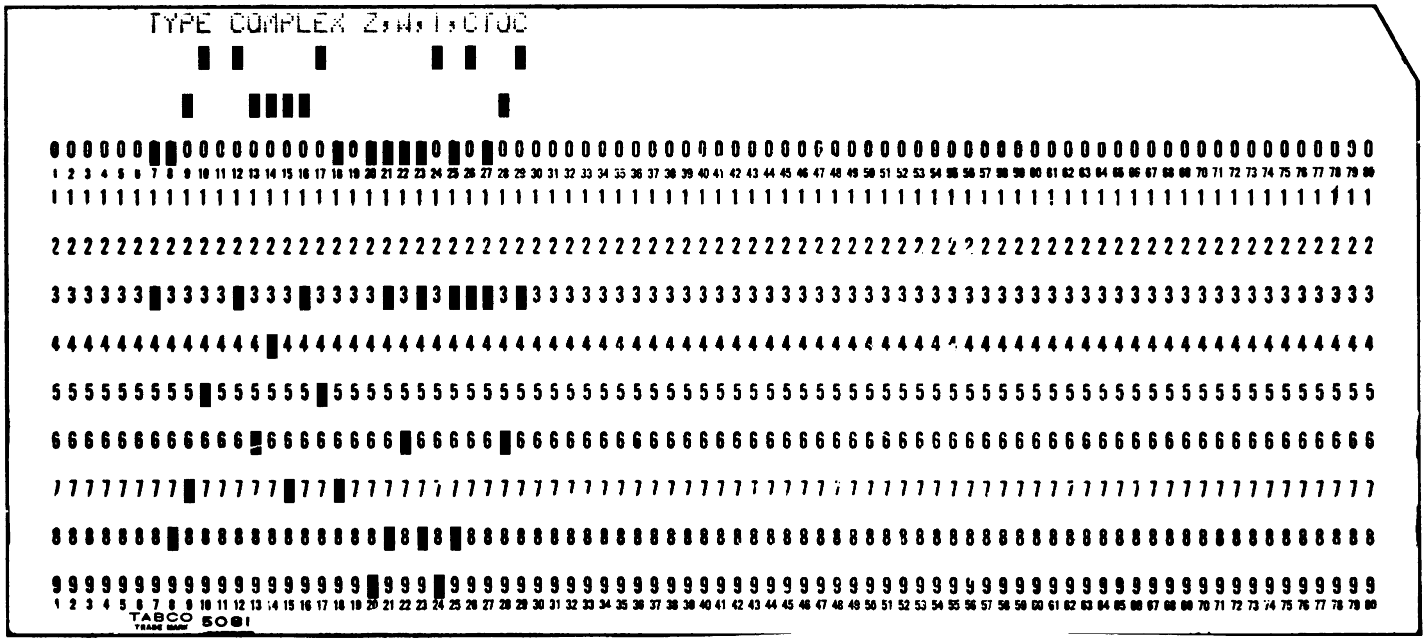
The FORTRAN*-63 language contains all of the features of its predecessor, FORTRAN-62, and forms an overset of the FORTRAN II language. The FORTRAN-63 compiler adapts current compiler techniques to the particular capabilities of the CONTROL DATA® 1604 and 3600 computer systems. Emphasis has been placed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

This reference manual was written as a text for advance 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.

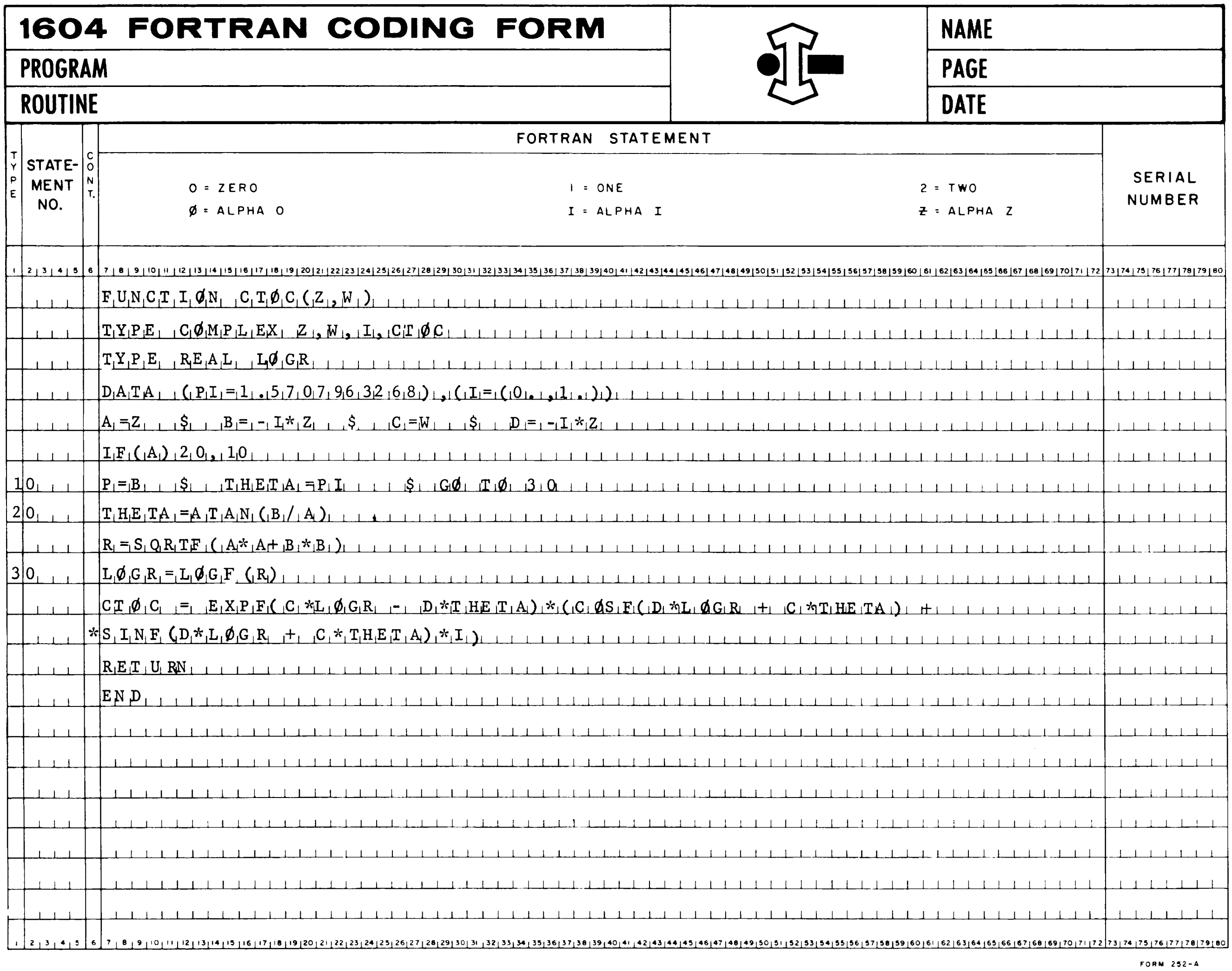
---

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

CONTENTS



Hollerith Card



1604 FORTRAN CODING FORM

PROGRAM

ROUTINE

NAME

PAGE

DATE

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT	SERIAL NUMBER

0 = ZERO
Ø = ALPHA O
1 = ONE
I = ALPHA I
2 = TWO
Ƶ = ALPHA Z

```
      FUNCTION CTØC(Z,W)
      TYPE CØMPLEX Z,W,I,CTØC
      TYPE REAL LØGR
      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=LØGF(R)
      CTØC = EXPF(C*LØGR - D*THETA)*(CØSF(D*LØGR + C*THETA) +
     *SINF(D*LØGR + C*THETA)*I)
      RETURN
      END
```

FORM 252-A

FORTRAN Coding Form

PROPERTIES AND ELEMENTS OF FORTRAN-63

1

1.1

CODING FORTRAN-63

CODING FORM FORTRAN-63 forms contain 80 columns in which the characters of the language are written, one character per column. Each line of the coding form corresponds to one 80-column punched card.

COMMENT CARD Comment information is designated by a C in column 1 of each line. Comment information will appear in the source program, but it is not translated into object code. Columns 2 through 80 may be used.

STATEMENT IDENTIFIERS Statements are identified by a string of up to five digits occupying any column positions, 1 through 5. Any statement may have an identifier, but only referenced statements require identification. Each statement identifier within a given program or subprogram must be unique.

Statement identifiers may range from 1 through 99999. Leading zeros are ignored; 1, 01, 001 are equivalent forms. Declarative statement identifiers (except FORMAT) are ignored by the compiler, except for diagnostic purposes.

STATEMENTS The statements of FORTRAN-63 are written in columns 7 through 72. Statements requiring more than one line may be carried to the next line by using a continuation designator. More than one statement may be written on a line. Blanks may be used freely in FORTRAN statements to provide readability. Blanks are significant, however, in Hollerith fields.

STATEMENT SEPARATOR $ The special character $ is used to write more than one statement on a line. Statements so written may also use the continuation feature. A $ symbol may not be used as a FORMAT statement separator.

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
```

CONTINUATION The first line of every statement must have a blank in column 6. If statements occupy more than one line of the coding sheet, all subsequent lines must have a non-blank, non-zero character in column 6. Any FORTRAN-63 statement may contain as many as 598 operators, delimiters (comma and parenthesis) and identifiers; blanks are not included in this count. Any number of continuations may extend a statement.

IDENTIFICATION FIELD Columns 73 through 80 are ignored in the translation process. These columns, therefore, may be used by the programmer for job identification and sequencing.

1.2

CONSTANTS

Four basic types of constants are used in FORTRAN-63: integer, octal, floating point and Hollerith. Complex and double precision constants can be formed from floating point constants. The type of a constant is determined by its form.

INTEGER Integer constants may consist of up to 15 decimal digits, in the range $0 \le n \le 2^{47}-1$. If the range is exceeded, the constant is treated as zero and a compiler diagnostic is provided.

Examples:

```
63              3647631

247             2

314159265       464646464
```

OCTAL

Octal constants may consist of up to 16 octal digits. The form is:

$n_1 \text{ --- } n_i B$

If the constant exceeds 16 digits, or if a non-octal digit appears, the constant is treated as zero and a compiler diagnostic is provided.

Examples:

7777777700000000B

7777700077777B

2323232323232323B

77B

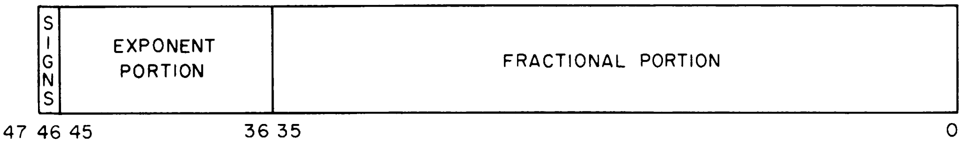
7777777777777700B

FLOATING POINT

Floating point quantities all have an exponent and a fractional part.

REAL

Word Structure



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

nE   n.n   n.   .n   nE±s   n.nE±s   n.E±s   .nE±s

n is the base value; s is the exponent to the base 10. The plus sign may be omitted for positive s. The range of s is 0 through 308.

If a plus or minus operator follows nE in an expression, the form (nE) or nEo must be used. If the range of a real constant is exceeded, the constant is treated as zero and a compiler diagnostic is provided.

Examples:

3.1415768

314.

.0749162

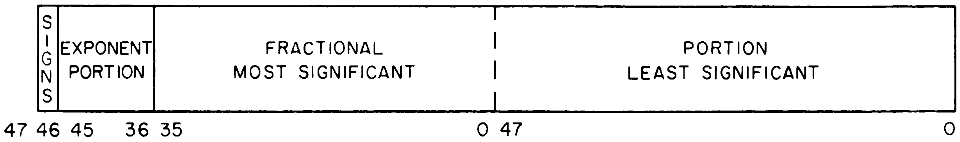
314159E-05

31.41592E-01

.31415E01

.31415E+01

DOUBLE Word Structure



Double precision constants are represented by a string of up to 25 digits. The forms are:

nD n.nD n.D .nD nD ± s n.nD ± s n.D ± s .nD ± s

n is the base value; s is the exponent to the base 10.

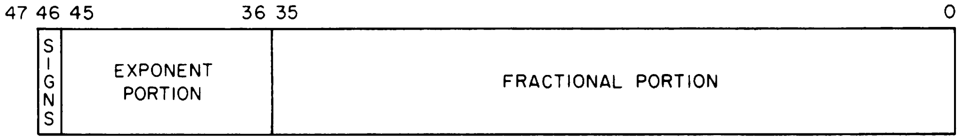
The D must always appear. The plus sign may be omitted for positive s; the range of s is 0 through 308. If the range is exceeded, the constant is treated as zero and a compiler diagnostic is provided.

Examples:

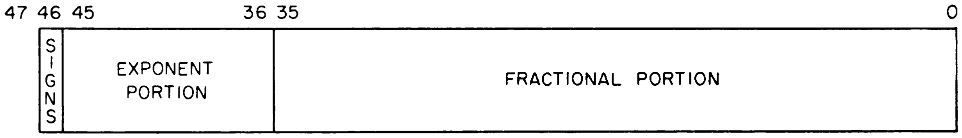
3.14159265358979323846 26D	31415.D-04
3.1415D	379867524430111D+01
3.1415D0	
3141.598D-03	

If a plus or minus operator follows nD, n.nD, n.D or .nD in an expression, the constant representation must be placed within parentheses or must be followed by a zero.

COMPLEX Generalized Word Structure



REAL VALUE



IMAGINARY VALUE

Complex constants are represented by pairs of real constants separated by a comma and enclosed in parentheses (R_1, R_2). R_1 represents the real part of the complex number and R_2, the imaginary part. Either constant may be preceded by a minus sign.

If the range of the reals comprising the constant is exceeded, a compiler diagnostic is provided. Diagnostics also occur when the number pair consists of integer constants, including (0,0).

Examples:

FORTRAN-63 Representation	Complex Number
(1., 6.55)	1. + 6.55i
(15., 16.7)	15. + 16.7i
(-14.09, 1.654E-04)	-14.09 + .0001654i
(0., -1.)	-i

HOLLERITH A Hollerith constant is a string of alphanumeric characters of the form hHf, h is an unsigned decimal integer between 1 and 120 characters representing the length of the field f. Spaces are significant in the field f. When h is not a multiple of 8, the last computer word is left-justified with BCD spaces filling the remainder of the word.

An alternate form of a Hollerith constant is hRf. When h is not a multiple of 8, the last computer word is right-justified with zero fill.

When h is greater than 120 only the first 120 characters are retained and the excess characters are discarded, but no diagnostic is provided.

Examples:

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

1.3
VARIABLES

FORTRAN-63 recognizes simple and subscripted variables. A simple variable represents a single quantity; a subscripted variable represents a single quantity within an array of quantities. The variable type is either defined in a TYPE declaration (section 4.1) or determined by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates a fixed point (integer) variable; any other first letter indicates a floating point (real) variable.

1.3.1

SIMPLE

A simple variable is the name of a storage area in which values can be stored. The variable is referenced by the location name; the value specified by the name is always the current value stored in that location.

SIMPLE INTEGER

variables are identified by 1 to 8 alphabetic or numeric characters; the first must be I, J, K, L, M, or N. Any integer value in the range from $-(2^{47} -1)$ to $2^{47} -1$ may be assigned to a simple integer variable.

Examples:

N	NOODGE
K2SO4	M58
LOX	M 58

Since spaces are ignored in variable names, M58 and M 58 are identical.

SIMPLE FLOATING POINT

variables are identified by 1 to 8 alphabetic or numeric characters; the first must be alphabetic and <u>not</u> I, J, K, L, M, or N. Any value from 10^{-308} to 10^{308} and zero can be assigned to a simple floating point variable.

Examples:

VECTOR	A65302
BAGELS	BATMAN

1.3.2

SUBSCRIPTED VARIABLE ARRAYS

An array is a block of successive memory locations which is divided into areas for storage of variables. Each element of the array is referenced by the array name plus a subscript. The type of an array is determined by the array name or TYPE declaration. Arrays may have one, two, or three dimensions and the maximum number of elements is the product of the dimensions. A subscript can be an integer constant, an integer variable, or any integer expression. Any other constant, variable, or expression will be reduced to an integer value. The array name and its dimensions must be declared at the beginning of the program in a DIMENSION statement (section 4.2).

ARRAY STRUCTURE Elements of arrays are stored by columns in ascending order of storage location. In the array declared as A(3,3,3):

$$\begin{array}{|ccc|} \hline A_{111} & A_{121} & A_{131} \\ A_{211} & A_{221} & A_{231} \\ A_{311} & A_{321} & A_{331} \\ \hline \end{array}$$

$$\begin{array}{|ccc|} \hline A_{112} & A_{122} & A_{132} \\ A_{212} & A_{222} & A_{232} \\ A_{312} & A_{322} & A_{332} \\ \hline \end{array}$$

$$\begin{array}{|ccc|} \hline A_{113} & A_{123} & A_{133} \\ A_{213} & A_{223} & A_{233} \\ A_{313} & A_{323} & A_{333} \\ \hline \end{array}$$

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

$$\begin{array}{lll} A_{111} \longrightarrow L & A_{121} \longrightarrow L+3 \; \ldots & A_{133} \longrightarrow L+24 \\ A_{211} \longrightarrow L+1 & A_{221} \longrightarrow L+4 \; \ldots & A_{233} \longrightarrow L+25 \\ A_{311} \longrightarrow L+2 & A_{321} \longrightarrow L+5 \; \ldots & A_{333} \longrightarrow L+26 \end{array}$$

If more than three subscripts appear, a compiler diagnostic is given. Program errors may result if subscripts are larger than the dimensions initially declared for the array. A single subscript notation may be used for a two or three dimensional array if it does not exceed the product of the declared dimensions.

1.3.3

SUBSCRIPT FORMS

A standard subscript has one of the following forms; c and d are unsigned integer constants and I is a simple integer variable:

(c * I ± d)

(I ± d)

(c * I)

(I)

(c)

A non-standard subscript is any arithmetic expression, other than the standard forms, used as a subscript.

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)

The location of an array element with respect to the first element is a function of the maximum array dimensions and the type of the array. 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

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

The quantity in braces is the subscript expression. If it is not an integer value, it is truncated after evaluation.

E is the element length, that is, the number of storage words required for each element of the array; for real and integer arrays, E = 1.

1. Referring to the matrix in 1.2.2 the location of A (2,2,3) with respect to A (1,1,1) is

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

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:

   $$\begin{aligned} z &= \text{Locn}\ \{Z(1,1,1)\} + \{2-1+5(1-1+5(4.67)\ )\}\ \text{Integer part} \\ &= \text{Locn}\ \{Z(1,1,1)\} + \{117.75\}\ \text{Integer part} \\ &= \text{Locn}\ \{Z(1,1,1)\} + 117 \end{aligned}$$

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

Given $A(D_1,D_2,D_3)$
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)

and for $A(D_1)$

A implies A(1)

However, the elements of a single-dimension array $A(D_1)$ may not be referred to as A(I,J,K) or A(I.J). Diagnostics will occur if this is attempted.

Array allocation is discussed under Storage Allocation in Chapter 4.

SUBSCRIPTED INTEGER VARIABLES,

the elements of an integer array, can be assigned the same values as simple integer variables. An integer array is named by an integer variable name (1 to 8 alphabetic or numeric characters the first of which is I, J, K, L, M, or N).

NEURON (6, 8, 6)
MORPH (20,20)
L6034(J, 3)
N3 (1)

SUBSCRIPTED FLOATING POINT VARIABLES,

the elements of a floating point array, can be assigned the same values as simple floating point variables. A floating point array is named with a floating point variable name (1 to 8 alphabetic or numeric characters, the first of which is alphabetic and not I, J, K, L, M, or N).

Examples:

TMESIS (6, 4, 7)	YCLEPT (46)
PST (20, 3, 3)	SVELTE (6, 8)

1.4
STATEMENTS

The FORTRAN-63 elements are combined to form statements. 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.

1.5
EXPRESSIONS

An expression is a constant, variable (simple or subscripted), function (section 7.2) or any combination of these separated by operators and parentheses, written to comply with the rules given for constructing a particular type of expression.

There are four kinds of expressions in FORTRAN-63: arithmetic and masking (Boolean) expressions which have numerical values, and logical and relational expressions which have truth values. For each type of expression there is an associated group of operators and operands.

ARITHMETIC EXPRESSIONS AND REPLACEMENT STATEMENTS

2

2.1

ARITHMETIC REPLACEMENT STATEMENTS

The general form of the arithmetic replacement statement (arithmetic statement) is A = E, where E is an arithmetic expression and A is any variable name, simple or subscripted. The operator = means that A is replaced by the value of the evaluated expression, E, with conversion for mode if necessary.

2.2

ARITHMETIC EXPRESSIONS

An arithmetic expression can contain the following operators:

Symbol	Function
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Operands are: Constants
Variables (simple or subscripted)
Functions (Chapter 7)

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
```

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 algebraic arithmetic expressions.

Rules:

1 An arithmetic expression may not contain adjacent arithmetic operators: x op op Y

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

3 If X, Y are expressions, then the following are expressions:

X + Y   X - Y   X/Y   X * Y

4 Expressions of the form X**Y and X**(-Y) are legitimate, subject to the restrictions in section 2.3.

5 The following forms of implied multiplication are permitted:

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

Complex constants are not included in implied multiplication:
constant (. . .) does not imply constant * (. . .)

2.2.1

ORDER OF EVALUATION

Hierarchy of arithmetic operation:

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

In an expression with no parentheses or within a pair of parentheses, in which unlike classes of operators appear, evaluation proceeds in the above order. In those expressions where operators of like classes appear, evaluation proceeds from left to right. For example, A**B**C is evaluated as (A**B)**C.

In parenthetical expressions within parenthetical expressions, evaluation begins with the innermost expression. Parenthetical expressions are evaluated as they are encountered in the left to right scanning process.

> When writing an integer expression it is important to remember not only the left to right scanning process, but also that dividing an integer quantity by an integer quantity always yields a truncated result; thus 11/3 = 3. The expression I*J/K will yield a different result than the expression J/K*I. For example, 4*3/2=6 but 3/2*4 =4.

When an integer expression contains parenthetical expressions with * or / operators, it is important to remember that the compiler will evaluate as many operations after a parenthetical expression as possible until it must do an intermediate or final store.

Example:

1. Z=X-Y+A/B*(C+D)*E is evaluated as
   Z=(C+D)*A/B*E+X-Y without an intermediate store.
2. Z=X-Y+A*B/(C+D)*E is evaluated as
   Z=A*B/(C+D)*E+X-Y with an intermediate store for (C+D).

Examples:

In the following examples, R indicates an intermediate result in evaluation:

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:

$Z \longrightarrow R_1$
$COSF(R_1) \longrightarrow R_2$
$R_2**2 \longrightarrow R_3$
$R_3*C(I,J+2) \longrightarrow R_4$
$R_4+H(13) \longrightarrow R_5$ 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

2.3

MIXED MODE ARITHMETIC EXPRESSIONS

FORTRAN-63 permits full mixed mode arithmetic. Mixed mode arithmetic is accomplished through the special library subroutines. In the 1604 computer system, these routines include double precision and complex arithmetic. The five standard operand types are complex, double, real, integer, and logical.

The programmer may also define three non-standard types. TYPE Declarations are covered in section 4.1 for standard types and Chapter 5 for non-standard types.

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.

Rules:

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

COMPLEX
DOUBLE
REAL
INTEGER
LOGICAL

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

3 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 1.

4 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, unless B is an integer constant less than 9.
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 B

Type A	I	R	D	C
I	I	R	D	C
R	R	R	D	C
D	D	D	D	C
C	C	C	C	C

mode of A**B

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 of non-standard multi-word type, the programmer must provide subroutines for the evaluation of A**B.

2.3.1 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:

$I\text{-}J \longrightarrow R_1$	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	
$A1*R_1 \rightarrow R_2$	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:

$A1\text{-}A2 \rightarrow R_1$	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

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

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

2.4

MIXED MODE REPLACEMENT STATEMENT

The mode of an evaluated expression is determined by the type of the dominant operand. This, however, does not restrict the types that identifier A may assume. An expression of complex mode may replace A even if A is type real. The following chart shows the A, E relationship for all the standard modes.

ARITHMETIC REPLACEMENT STATEMENT $A = E$

A is an Identifier E is an Arithmetic Expression

$\phi(f)$ is the Evaluated Arithmetic Expression

Mode of $\phi(f)$ TYPE of A	Complex	Double	Real	Integer
Complex	Store real & imaginary parts of $\phi(f)$ in real & imaginary parts of A.	Round $\phi(f)$ to real. Store in real part of A. Store zero in imaginary part of A.	Store $\phi(f)$ in real part of A. Store zero in imaginary part of A.	Convert $\phi(f)$ to real & store in real part of A. Store zero in imaginary part of A.
Double	Discard imaginary part of $\phi(f)$ & replace it with ± 0 according to real part of $\phi(f)$.	Store $\phi(f)$ (most & least significant parts) in A (most & least significant parts).	If $\phi(f)$ is ± affix ± 0 as least significant part. Store in A, most & least significant parts.	Convert $\phi(f)$ to real. Fill out least significant half with binary zeros or ones accordingly as sign of $\phi(f)$ is plus or minus. Store in A, most and least significant parts.
Real	Store real part of $\phi(f)$ in A. Imaginary part is lost.	Round $\phi(f)$ to real & store in A. Least significant part of $\phi(f)$ is lost.	Store $\phi(f)$ in A.	Convert $\phi(f)$ to real. Store in A.
Integer	Truncate real part of $\phi(f)$ to INTEGER. Store in A. Imaginary part is lost.	Truncate $\phi(f)$ to INTEGER & store in A.	Truncate $\phi(f)$ to INTEGER. Store in A.	Store $\phi(f)$ in A.
Logical	If real part of $\phi(f)\neq 0$, $1 \longrightarrow A$. If real part of $\phi(f)=0$, $0 \longrightarrow A$.	If $\phi(f)\neq 0$, store 1 in A. If $\phi(f)=0$, store 0 in A.	Same as for double at left.	Same as for double at left.

When all of the operands in the expression E are of type logical, the expression is evaluated as if all the logical operands were integers.

For example, if L_1, L_2, L_3, L_4 are logical variables, R is a real variable, and I is an integer variable, then

$$I = L_1 * L_2 + L_3 - L_4$$

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) $A_1 = C_1 * C_2 - C_3 / C_4$ (.905, 15.393) = (4.4, 2.1) * (3.0, 2.0) - (3.3, 6.8)/(1.1, 3.4)

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) $A_3 = C_1$ 4.4000E 00 = (4.4, 2.1)

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) $A_3 = C_1 * (0.,-1.)$ 2.1000E 00 = (4.4, 2.1)*(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) $A_4 = R_1 / R_2 * (R_3 - R_4) + I_1 - (I_2 * R_5)$ 3 = 8.4/4.2 * (3.1-2.1) + 14 - (1*2.3)

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) $A_2 = D_1{*}{*}2{*}(D_2+(D_3{*}D_4))$ $+(D_2{*}D_1{*}D_2)$

4.96800000000000000000000E 01=
2.0D**2*(3.2D+(4.1D*1.0D))
+(3.2D*2.0D*3.2D)

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) $A_5 = C_1{*}R_1-R_2+I_1$

1 = (4.4, 2.1) * 8.4 - 4.2 + 14

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.

LOGICAL/RELATIONAL AND MASKING EXPRESSIONS AND REPLACEMENT STATEMENTS

3

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

3.1

LOGICAL EXPRESSION

A logical expression has the general form

$$O_1 \text{ op } O_2 \text{ op } O_3 \ldots$$

The terms O_i are logical variables, arithmetic expressions or relational expressions, and op is the logical operator .AND. indicating conjunction or .OR. indicating disjunction.

The logical operator .NOT. indicating negation appears in the form:

$$.\text{NOT}.\ O_1$$

The value of a logical expression is either true or false.

When an arithmetic expression appears as a term of a logical replacement statement, the value of the expression is examined. If the value is nonzero, the term has the value TRUE. If the value is equal to zero, the term has the value FALSE.

Logical expressions are generally used in logical IF-statements. (See section 6.3)

Rules:

1. The hierarchy of logical operations is:

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

2. 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.

3 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.

4 .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. · · · Other combinations will cause compilation diagnostics.

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

.NOT. $\mathcal{L}_1$	is false if and only 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

Incorrect usages such as the following will cause compiler diagnostics.

A.GT.(B.AND.C)

10.LE.N.LE.100

Q.NOT. .OR.R

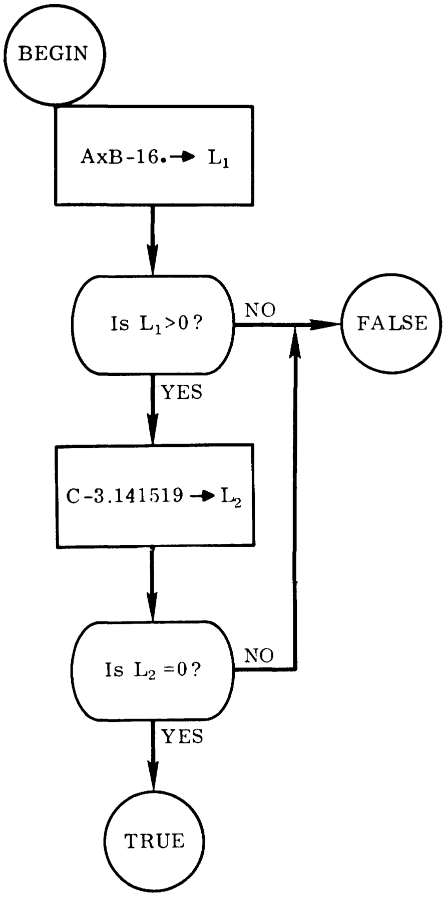
C.AND. .NOT. .NOT.B

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

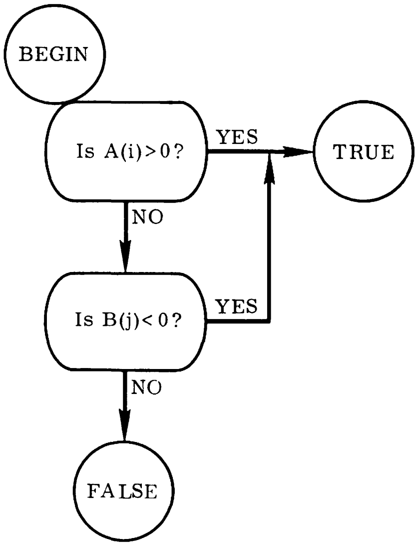
Examples:

Logical Expressions

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

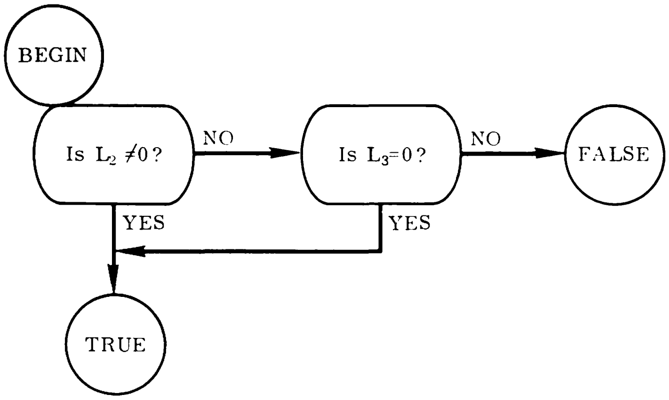


{A(I) greater than 0} .OR. {B(J) less than 0}
A(I) .GT. 0 .OR. B(J) .LT. 0

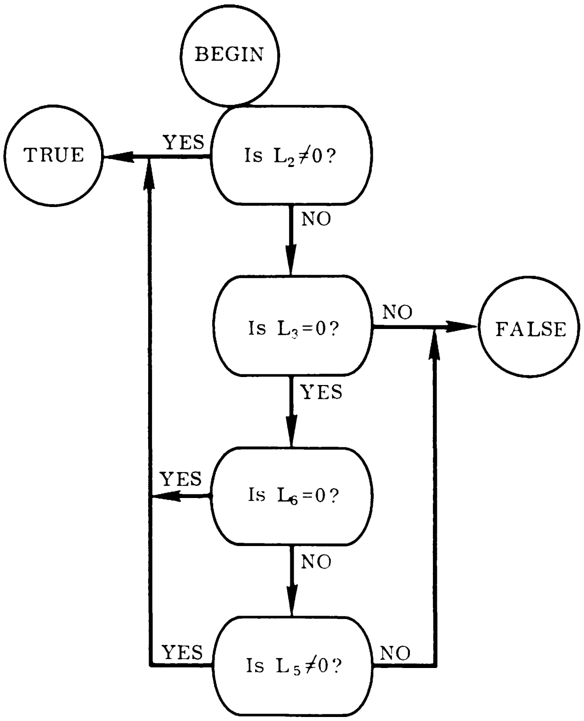


In the two examples below, all L_i are of TYPE LOGICAL

(L2 .OR. .NOT. L3)



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



3.2

RELATIONAL EXPRESSION

A relational expression has the form:

q_1 op q_2

The q's are arithmetic expressions; op is an operator belonging to the set:

Operator	Meaning
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to

A relation is true if q_1 and q_2 satisfy the relation specified by op. A relation is false if q_1 and q_2 do not satisfy the relation specified by op.

Relations are evaluated as illustrated in the relation, p .EQ. q. This is equivalent to the question, does $p-q = 0$?

The difference is computed and tested 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.

Rules:

1 The permissible forms of a relation are:

q_1 op q_2	
q	by itself, in which case a non-zero value is true and a zero value is false.

2

q_1 op q_2 op q_3. . .	is <u>not</u> permissible
q_1 op q_2 .AND. q_2 op q_3 . . .	is the correct form

3 The evaluation of a relation of the form q_1 op q_2 is from left to right. The relations q_1 op q_2, q_1 op (q_2), (q_1) op q_2, (q_1) op (q_2) are equivalent.

Examples:

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

3.3

MASKING REPLACEMENT STATEMENT

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

1. The type of M must be real or integer.
2. All operands in the expression E must be type real or integer. E may contain functions as well as variable or constant operands.

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. FUNC (X,T)
```

3.4

MASKING EXPRESSIONS

In a FORTRAN-63 masking expression 48-bit arithmetic is performed bit-by-bit on the operands within the expression. The operands must be type real or integer only. Type integer includes octal and Hollerith constants. If operands of other types are used, a diagnostic will occur.

Although the masking operators are identical in appearance to the logical operators, their meanings are different. They are listed according to hierarchy, and the following definitions apply:

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

Rules:

1 Let B_i be variables or constants whose types are real or integer or masking expressions. Then the following are masking expressions.

.NOT. B_1

B_1 .AND. B_2

B_1 .OR. B_2

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

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

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

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

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

5 Masking expressions must not contain parenthetical arithmetic expressions or statement functions.

6 A masking expression in a logical IF statement is interpreted as a logical expression. The appearance of a masking expression in an arithmetic IF will cause a diagnostic.

Examples:

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

.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

3.5 MULTIPLE REPLACEMENT STATEMENTS

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

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

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 and:

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:

A	real
E,F	complex
G	double
I	integer
K	logical

The numbers in the examples represent the evaluations of expressions.

A = G = 3.1415926535897932384626D
3.1415926535897932384626D → G
3.141592654 → A

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 ALLOCATIONS

4

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

4.1

TYPE DECLARATIONS

The TYPE declaration provides the compiler with information on the structure of variable and function identifiers. There are five standard variable types (non-standard types are explained in Chapter 5). Type is declared by one of the following statements:

Statement	Characteristics	
TYPE COMPLEX List	2 words/element	Floating point
TYPE DOUBLE List	2 words/element	Floating point
TYPE REAL List	1 word/element	Floating point
TYPE INTEGER List	1 word/element	Integer
TYPE LOGICAL List	1 word/element	Logical (non-dimensioned)
	32 elements/word	Logical (dimensioned)

A list is a string of identifiers separated by commas; subscripts are not permitted. An example of a list is:

A, B1, CAT, D36F, EUPHORIA

Rules:

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

2 If an identifier is declared in two or more TYPE declarations, a compilation diagnostic will occur.

3 An identifier not declared in a TYPE statement will be an integer if the first letter of the identifier is I, J, K, L, M, N; for any other letter, it will be real.

Examples:

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

4.2

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 $V_1, V_2, \ldots, V_n$

The variable names, V_i, may have 1, 2, or 3 integer constant subscripts separated by commas, as in SPACE (5, 5, 5). Under certain conditions within subprograms only, the subscripts may be integer variables. This is explained in section 7.11.1.

The number of computer words reserved for a given array is determined by the product of the susbcripts in the subscript string, and the type of the variable. A maximum of 2^{15} -1 elements may be reserved in any given array. In the statements

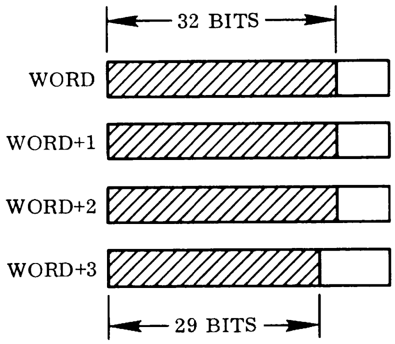
```
TYPE COMPLEX HERCULES
DIMENSION HERCULES (10, 20)
```

the number of elements in the array HERCULES is 200. Two words are used to store a complex 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)
```

the 125 elements in the array XERXES will occupy four sequential words as shown below.



4.2.1
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 must not exceed the maximum array size specified by the DIMENSION statement in the calling program. See section 7.11 for details and examples.

4.3
COMMON

A program may contain or call subprograms. Areas of common information may be specified by the statement:

COMMON /I_1/ List / I_2/ List . . .

I is a common block identifier up to 8 characters in length which designates either labeled or numbered common block. If the first letter is alphabetic, the identifier denotes a labeled common block; the remaining characters may be alphabetic or numeric. If the first letter is numeric, the remaining characters must be numeric and the identifier denotes a numbered common block. Leading zeros in numeric identifiers are ignored. Zero by itself is an acceptable numbered common block identifier. The following are common identifiers:

Labeled	Numbered
AZ13	1
MAXIMUS	146
Z	3600
XRAY	0

List is composed of simple variable identifiers and array identifiers (subscripted or non-subscripted). If a non-subscripted array name appears in the list, the dimensions are defined by the DIMENSION statement in that program.

Arrays may also be dimensioned in the COMMON statement when a subscript string appears with the identifier. If dimensioned in both, those in the DIMENSION statement will be used and an informative diagnostic will be given. Execution will not be deleted.

The common block identifier with or without the separating slashes may be omitted for blank common. Blank common is treated as numbered common by the compiler.

Examples:

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

4.4
COMMON BLOCKS

The COMMON statement provides the programmer with a means of reserving blocks of storage area that can be referenced by more than one subprogram. The statement reserves both numbered and labeled blocks. Only labeled common blocks may be preset; that is, data may be stored in labeled common blocks by the DATA statement and is made available to any subprogram using the appropriate labeled block.

If a subprogram does not use all of the locations reserved in a common block, unused variables may be necessary in the COMMON statement to insure proper correspondence of common areas.

MAIN PROG	COMMON/SUM/A, B, C
SUB PROG	COMMON/SUM/E, F, G

In the above example, only the variables E and G are used in the subprogram. The unused variable F is necessary to space over the area reserved by B.

Rules:

1 COMMON is non-executable and must precede the first executable statement in the program. Any number of COMMON statements may appear in a program section.

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

3 Labeled common block identifiers are used only for block identification within the compiler; they may be used elsewhere in the program as other kinds of identifiers.

4 An identifier in one common block may not appear in another common block. If it does the identifier is doubly defined.

5 The order of the arrays in a common block are determined by the COMMON statement.

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

Violations of rules 1 and 4 result in compiler diagnostics.

4.4.1
BLOCK LENGTH

The length of a common block in computer words is determined from the number and type of the list identifiers. In the following statements, the length of the common block A is 12 computer words. The origin of the common block is Q(1). (Q and R are real variables and S is complex).

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

	block A	
origin	Q(1)	
	Q(2)	
	Q(3)	
	Q(4)	
	R(1)	
	R(2)	
	R(3)	
	R(4)	
	S(1)	real part
	S(1)	imaginary part
	S(2)	real part
	S(2)	imaginary part

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
DIMENSION A(10)
COMMON A
```

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

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

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

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

The label of a COMMON block is used only for block identification. The following is permissible:

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

4.5

EQUIVALENCE

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

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

(A,B, . . .) is an equivalence group of two or more 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) is the same as A(the value of (i+I((j-1)+J(k-1)))

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

Example:

EQUIVALENCE is most commonly used when two or more arrays can share the same storage locations. The lengths may be different or equal.

```
      DIMENSION A(10,10), I(100)
      EQUIVALENCE (A,I)
                    .
                    .
                    .
  5   READ 10, A
                    .
                    .
                    .
  6   READ 20, I
```

The EQUIVALENCE statement assigns the first element of array A and array I to the same storage location. The READ statement 5 stores the A array in consecutive locations. Before statement 6 is executed all operations using A should be completed as the values of array I will be read into the storage locations previously occupied by A.

Rules:

1 EQUIVALENCE is non-executable and must precede the first executable statement in the program or subprogram.

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

3 Any full or multi-word variable, standard or non-standard type, may be made equivalent to any other full or multi-word variable. The variables may be with or without subscript.

Any partial word variable, standard logical or non-standard byte, may be made equivalent to any type of partial, full, or multi-word variable. The partial word variable must be unsubscripted.

4 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
Sa = subscript of A
Sb = subscript of B

<u>CASE I A, B both in COMMON</u>

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

$Sa \geq Sb$ is a permissible subscript arrangement
$Sa < Sb$ is not

b) If B appears before A in the COMMON statement

$Sa \leq Sb$ 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 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)

$Sb \leq Sa$ 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)

$Sa \leq Sb$ 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.

4.6
DATA

The programmer may assign constant values to variables in the source 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.

DATA(I_1=List), (I_2=List), . . .

I is an identifier representing a simple variable, array name, or a variable with integer constant subscripts or integer variable subscripts.

List contains constants only and has the form

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

k is an integer constant repetition factor that causes the parenthetical list following it to be repeated k times. If k is non-integer, a compiler diagnostic occurs.

Rules:

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

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

3. DO loop-implying notation is permissible with the restriction that the third indexing parameter, m_3 cannot appear. This notation may be used for storing constant values in arrays.

```
DIMENSION GIB (10)

DATA ((GIB(I),I=1,10)=1. ,2. ,3. ,7(4.32))

     ARRAY GIB    1.
                  2.
                  3.
                  4.32
                  4.32
                  4.32
                  4.32
                  4.32
                  4.32
                  4.32
```

4. Variables in blank or numbered common or variable dimensioned arrays may not be preset in a DATA statement. Violation of this rule causes an assembly listing C error.

5. Either unsigned constants or constants preceded by a minus sign may be used. Octal constants prefixed with minus signs will be stored in complement form; use of .NOT. will cause a compiler diagnostic.

6. In the DATA statement, the type of the constant stored is determined by the structure of the constant rather than by the identifier in the statement. In DATA (A=2), an integer 2 replaces A, not a real 2 as might be expected from the form of the identifier.

7. There should be a one-one correspondence between the identifiers and the list. This is particularly important in arrays. For instance

```
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, errors may occur when B is referred to elsewhere in the program.

```
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.

When the number of list elements exceeds the range of the implied DO, the excess list elements are stored in consecutive locations starting with the first location specified in the DO-loop.

```
DATA ((A(I), I=1,5) =1. , . . . , 10.)
```

The excess values 6 through 10. are stored in locations A through A + 4.

8 Non-standard type variables are permitted. However, for a byte size variable, the constant value in the list must fill the entire computer word.

```
        .
        .
        .
TYPE OTHER5 (/6)A

DIMENSION A(8)

DATA (A=4142434445464761B)
```

9 Use of DATA with a logical variable constitutes a special case, as shown in the following example.

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 by columns.

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)
```

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
```

TYPE-OTHER DECLARATION 5

FORTRAN-63 allows eight distinct modes of arithmetic. The mode and the size of the operand is fixed for the five standard types — real, integer, double, complex and logical (TYPE Declarations, 4.1). The routines or instructions required to handle these arithmetic modes are provided with the system.† For further detail see Appendix E, part A.

The programmer can define up to three modes of non-standard arithmetic arbitrarily identified as types 5, 6, 7. A non-standard type is arbitrary both in mode and execution and may specify multi-word elements (operands) or partial word elements, called bytes.

The mode and structure of the operand is defined in the TYPE-other declaration. Execution of all expressions containing non-standard variables must be defined in routines supplied by the user (Appendix E, part B).†† Examples of non-standard operations with user routines are given at the end of Appendix E.

Non-standard types may be used to introduce a new type of arithmetic by giving new meaning to the basic arithmetic operators. In a standard arithmetic expression, a + symbol has the fixed interpretation "to add". In a non-standard expression, the programmer may, for example, define + to mean "shift" or "cube".

Non-standard types also may be used to extend precision up to seven computer words or may specify only part of a word in arithmetic operations.

---

† The following exponentiation routines are provided:

real**real
real** integer
real**double
real** complex

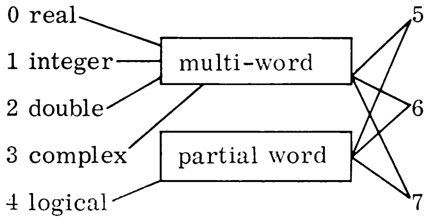
integer**integer
integer**double
integer**complex
integer**real

double**double
double**complex
double**real
double**integer
complex**integer

complex**complex
complex**real
complex**double
} These exponentiation routines will give an error message when called.

†† For exponentiation, if the exponent is an integer constant 1-8, the value is calculated by successive multiplications which may or may not be calculated in a separate subroutine.

	Standard	Non-Standard
Number of Types	5	3
Arithmetic Mode and Element Structure	Fixed	Arbitrary (defined in TYPE-other declarations)
Arithmetic operations	fixed (defined in system routines)	arbitrary (defined in user-provided routines)



The steps in solving a non-standard operation are:

1. Define a problem
2. Write and compile a program to solve the problem
    a. Define non-standard variables in TYPE-other declarations (Chapter 5)
3. Analyze the calls to subroutines generated by the compiler (Appendix E, Part A)
4. Provide a subroutine with the calls as entry points; the subroutine will perform the operations desired by the programmer (Appendix E, Part B
5. Compile and execute the program and subroutine (Chapter 10, Deck Structure

5.1
TYPE-OTHER DECLARATIONS

The TYPE-other declaration provides the compiler with information regarding the structure of the non-standard identifier that names variables and functions.

The general form of a non-standard declaration is:

TYPE name# (/b) List

or

TYPE name# (w) List

name# is an arbitrary alphanumeric identifier, 2-8 characters. The last character, #, must be one of the type indicators 5, 6, or 7.

(/b) specifies the number of bits in a partial word element. b must be a divisor of 48; if it is not, a compilation diagnostic will be given.

TYPE BYTE5 (/6) A	A is a 6-bit element
TYPE PARTS6 (/3) MAX	MAX is a 3-bit element

(w) specifies the number of words in a multi-word element. w must be in the range 1-7; otherwise, a compilation diagnostic will be given.

TYPE DOUBLE7 (4) OX	OX is a 4-word element

List is a string of simple variable identifiers, or array names, separated by commas. Identifiers have w words per element or b bits per element. Both multi-word element and partial word element identifiers may be dimensioned in DIMENSION or COMMON statements.

An identifier is doubly defined if it occurs in more than one TYPE-other declaration:

TYPE BYTE5 (3) A,B
TYPE BYTE6 (/2) A, B
(this causes a compilation diagnostic.)

When simple partial word elements are specified, the leftmost b characters of a word are used. When partial word element arrays are specified, the elements are in consecutive locations, left to right, in the word. The number of elements in a word is 48/b.

Example:

DIMENSION A (13)

TYPE BYTE5 (/8) A, B

B						word 1
8 bits						
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	word 2
A(7)	A(8)	A(9)	A(10)	A(11)	A(12)	word 3
A(13)						word 4

A program may contain a maximum of three non-standard types (type 5, 6, 7). Two or more TYPE-other declarations of the same name and type with multi-word elements of different lengths may appear in the same program.

Examples:

TYPE SAM5 (6) A,B TYPE SAM5 (3) C,D	will compile correctly; the programmer must provide a way to determine the element length of variables which are the same type.
TYPE LIEBE6 (6) E,F TYPE LIEBE6 (/5) G,H	will cause a compilation diagnostic; only full word elements may be used.
TYPE PATTI7 (1) M TYPE BARBI7 (3) B	will cause a compilation diagnostic; the name must be the same.

5.2

EVALUATION OF NON-STANDARD ARITHMETIC EXPRESSIONS

1. The translation of a non-standard arithmetic expression by FORTRAN-63 follows the same rules of precedence as for standard arithmetic expressions: exponentiation, multiplication-division, addition-subtraction.

2. The scanning order of the expression is left to right.

3. The non-standard types (5, 6, 7) may not be mixed within an expression. Non-standard variables of the same type but with different element lengths may be mixed with each other.

4. Any one of the types 5, 6, 7 may be mixed with any of the standard types in arithmetic expressions.

5. The non-standard type dominates the mode of the evaluated expression.

6. A non-standard type specifying byte arithmetic may not participate in exponentiation unless the exponent is an integer constant 1-8.

7. If A or B or both are of non-standard multi-word type (and B is not an integer constant 1-8), the programmer must provide subroutines for the evaluation of A**B.

For further information on non-standard types in mixed mode arithmetic, see Mixed Mode Arithmetic Expression, in Chapter 2.

5.3

SAMPLE PROGRAM

The following is a simple example of what the programmer would encounter using non-standard variables in a non-standard arithmetic operation.

Step 1 Define problem
Add B to A by using a multiply operator, * . Store the value in C and print value in the form: C=

Step 2 Define variables
A and B are non-standard and are defined in the TYPE-other declaration:

TYPE OTHER5 (1) A,B

C is TYPE REAL

Step 3 Write a FORTRAN program and compile it

```
     PROGRAM OTHER
 2   TYPE REAL C
 3   TYPE OTHER5 (1) A,B
 4   A=4.1 $ B=5.4 $ C=A*B
 5   PRINT 1,C
 1   FORMAT (2HC=E14.8)
     END
```

Step 4 Analyze the calls to subroutines generated by the CODAP1 assembler.

```
PROGRAM         OTHER
                                  IDENT  OTHER
RANGE             FWA   -  LWA+1
                00000      00026
ENTRY POINTS
                00002      OTHER
EXTERNAL SYMBOLS
                00001   Q1Q00510
                00002   Q1Q10550
                00003   Q1Q00550
                00004   Q1Q04550
                00005   Q1Q10510
                00006   Q8QINGOT
                00007   Q8QENGOT
                00010   Q8QGOTTY
                00011   Q8QENTRY
00000+                  FORMAT.   BSS    2
                00002+            ENTRY  OTHER
00002+                  ENDING.   BSS    0
00002+  75  0   00002+  OTHER     SLJ    OTHER
        75  4   00023+  -         RTJ    INITIAL.
```

```
PROGRAM          OTHER

00003+  75  4  X00001  .4        CALL   Q1Q00510     Load accumulator with 4.1
        00  0   00024+            0     =02003406314631463
00004+  75  4  X00002  +         CALL   Q1Q10550     Store accumulator in A
        00  0   00021+            0     A
00005+  75  4  X00001  +         CALL   Q1Q00510     Load accumulator with 5.4
        00  0   00025+            0     =020035314631463 15
00006+  75  4  X00002  +         CALL   Q1Q10550     Store accumulator in B
        00  0   00020+            0     B
00007+  75  4  X00003  +         CALL   Q1Q00550     Load accumulator with A
        00  0   00021+            0     A
00010+  75  4  X00004  +         CALL   Q1Q04550     Multiply A by B
        00  0   00020+            0     B
00011+  75  4  X00005  +         CALL   Q1Q10510     Store product in C
        00  0   00022+            0     C
00012+  04  0   00000+ .5        ENQ    ..1
        10  0   00063            ENA    +51
00013+  75  4  X00006  +         RTJ    Q8QINGOT
        00  0   00000  -          0     0
00014+  75  4  X00010  +         RTJ    Q8QGOTTY
        00  0   00016+ -          0     GG00000.
00015+  00  0   00000             0     0
        01  0   00022+ -          1     C
00016+  75  4  X00007  GG00000.  RTJ    Q8QENGOT
        50  0   00000
                00000+           ORGR   FORMAT.
00000+  34  0   27063  ..1       BCD    2(2HC=E14.8)
        13  6   50104
00001+  73  1   07420
        20  2   02020
                00017+           ORGR   *
00017+  75  0   00002+           SLJ    ENDING.
        50  0   00000
00020+  00  0   00000  B         OCT    0
        00  0   00000
00021+  00  0   00000  A         OCT    0
        00  0   00000
00022+  00  0   00000  C         OCT    0
        00  0   00000
00006                            EXT    Q8QINGOT
00007                            EXT    Q8QENGOT
00010                            EXT    Q8QGOTTY
00011                            EXT    Q8QENTRY
00023+                 BEGIN.    BSS    0
00023+  75  0  X00011  INITIAL.  SLJ    Q8QENTRY
        75  0   00023+           SLJ    BEGIN.
00024+  20  0   34063
        14  6   31463
00025+  20  0   35314
        63  1   46315
                00000            END    OTHER
```

```
PROGRAM        OTHER

   NO DOUBLY DEFINED
   NO UNDEFINED SYMBOLS
   NO ASSEMBLY ERRORS
      NULLS                                   .4      .5
               SYMBOLIC REFERENCE TABLE       21         SYMBOLS
00021   A          00004   00007
00020   B          00006   00010
00023   BEGIN.     00023
00022   C          00011   00015
00002   ENDING.    00017
00000   FORMAT.    00017
00016   GG00000.   00014
00023   INITIAL.   00002
00002   OTHER      00002
00001   Q1Q00510   00003   00005
00003   Q1Q00550   00007
00004   Q1Q04550   00010
00005   Q1Q10510   00011
00002   Q1Q10550   00004   00006
00007   Q8QENGOT   00016
00011   Q8QENTRY   00023
00010   Q8QGOTTY   00014
00006   Q8QINGOT   00013
00003   .4
00012   .5
00000   ..1        00012
```

Step 5 Provide subroutines with the calls as entry points to perform the desired operation.

```
            IDENT          JOE

            ENTRY          Q1Q00510
Q1Q00510    SLJ            **
            LDA            *
+           ARS            24
            INA            -1
+           SAU            *+1

+           LDA      7     **
            SLJ            Q1Q00510
            ENTRY          Q1Q10550
Q1Q10550    SLJ            **
            STA            TEMP
+           LDA            *-1
            ARS            24
+           INA            -1
            SAL            *+1
```

```
+           LDA           TEMP
            STA       7   **
            SLJ           Q1Q10550
            ENTRY         Q1Q00550

Q1Q00550    SLJ           **
            LDA           *
+           ARS           24
            INA           -1
+           SAU           *+1

+           LDA       7   **
            SLJ           Q1Q00550
            ENTRY         Q1Q04550
Q1Q04550    SLJ           **
            STA           TEMP
+           LDA           *-1
            ARS           24
+           INA           -1
            SAL           *+1
+           LDA           TEMP
            FAD       7   **
            SLJ           Q1Q04550
            ENTRY         Q1Q10510

Q1Q10510    SLJ           **
            STA           TEMP
+           LDA           *-1
            ARS           24
+           INA           -1
            SAL           *+1

+           LDA           TEMP
            STA       7   **
            SLJ           Q1Q10510

TEMP        DEC

            END
```

CONTROL STATEMENTS 6

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 assembly the 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.

6.1 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 $1 \le N \le 99999$. An identifier up to 5 digits long may occupy any of the first five columns; blanks are squeezed out and leading zeros are ignored, 1, 01, 001, 0001, are identical.

Any statement label referenced in a control statement (with the exception of the Assigned GO TO) which does not appear as the label of an executable statement will appear in the category UNDEFINED SYMBOLS following the assembly listing. The number will be preceded by a period. If a reference is made to an unlabeled FORMAT statement, the label will appear as a number preceded by two periods.

If two or more executable statements have the same statement identifier, the label will appear in the category DOUBLY DEFINED following the assembly listing. The label will be preceded by a period. Doubly defined labels on FORMAT statements will appear as a number preceded by two periods.

Examples:

```
UNDEFINED SYMBOLS       .20       . .15
DOUBLY DEFINED          .399      . .3
```

6.2

GO TO STATEMENTS

Unconditional transfer of control is provided by GO TO statements.

UNCONDITIONAL GO TO

GO TO n

This statement causes an unconditional transfer to the statement labeled n; n is a statement identifier.

ASSIGNED GO TO

GO TO m, $(n_1,n_2, . . . ,n_m)$

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. Although a parenthetical list need not be present, it should appear when the statement is used in a DO-loop.

The comma after m is optional when the list is omitted. m cannot be the result of a computation. No compiler diagnostic is given if m is computed, but the object code will be incorrect.

ASSIGN STATEMENT

ASSIGN ℓ TO m

This statement is used with the Assigned GO TO statement. ℓ is a statement number, m is a simple integer variable.

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

Control will transfer to statement 10.

COMPUTED GO TO

GO TO $(n_1,n_2, . . . ,n_m)$i

GO TO $(n_1,n_2, . . . ,n_m)$, 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. Otherwise, transfer is to n_i.

For proper operations, i must not be specified by an ASSIGN statement. No compilation diagnostic is given for this error, but the object code will be incorrect.

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

Control will transfer to statement 21.

6.3 IF STATEMENTS

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

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 accordingly.

$A < 0$	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
```

TWO BRANCH IF (LOGICAL)

IF (L) n_1,n_2
L is a logical, relational, or arithmetic expression or any legal combination of the three. A masking expression will be interpreted as logical. 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
```

SENSE LIGHT

SENSE LIGHT i

The statement turns on the sense light i. SENSE LIGHT 0 turns off all sense lights. i may be a simple integer variable or constant (1 to 4).

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
```

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 1604 $1 \le i \le 48$ (CO OP Monitor function)

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

6.4

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

The above statements are equivalent. 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 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.

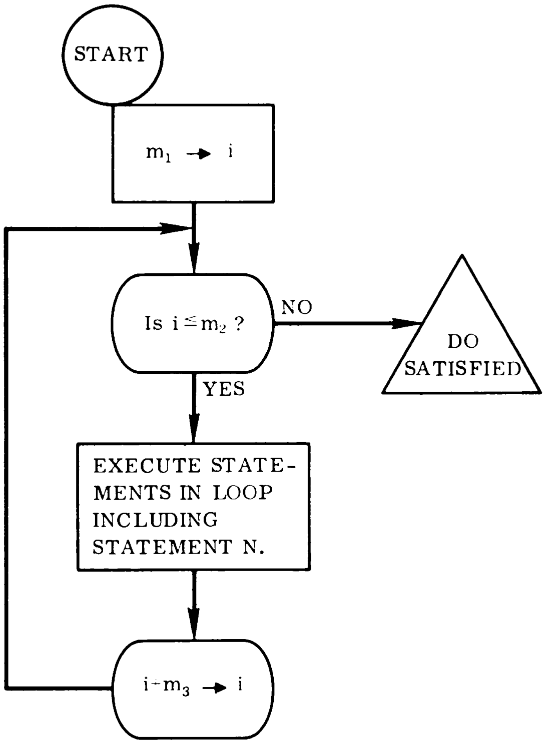
6.5
DO STATEMENT

DO n i = m_1,m_2,m_3
This statement makes it possible to repeat groups of statements and to change the value of a fixed point variable during the repetition. n is the number of the statement ending the DO loop. i is the index variable (simple integer). The m_i are the indexing parameters; they may be unsigned integer constants or simple integer variables. The initial value assigned to i is m_1, m_2 is the largest value assigned to i, and m_3 is the amount added to i after each DO loop is executed. If m_3 does not appear, it is assigned the value 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. See Transmission of Arrays section and DATA Statement section for usage of implied DO loops.

6.5.1
DO LOOP EXECUTION

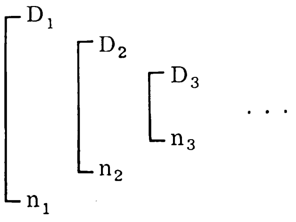
The initial value of i, m_1, is compared with m_2 before executing the DO loop and, if it does not exceed m_2, the loop is executed. After this step, i is increased by m_3. i is again compared with m_2; this process continues until i exceeds m_2 as shown below. Control then passes to the statement immediately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

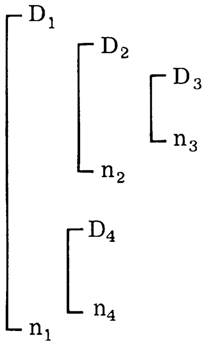
6.5.2 DO NESTS

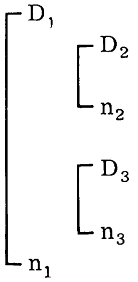
When a DO loop contains another DO loop, the grouping is called a DO nest. The last statement of a nested DO loop must either be the same as the last statement of the outer DO loop or occur before it. If $D_1, D_2, \ldots D_m$ represent DO statements, where the subscripts indicate that D_1 appears before 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.

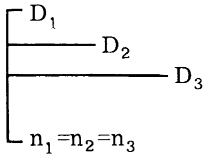


Examples:

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







```
  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
```

6.5.3

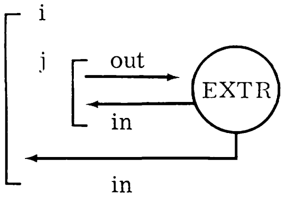
DO LOOP TRANSFER

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.



6.5.4

DO PROPERTIES

1) The indexing parameters m_1,m_2,m_3 are either unsigned integer constants or simple integer variables. Subscripted variables and negative or zero integer constants will cause a diagnostic.

2) The values of m_2 and m_3 may be changed during the execution of the DO loop.

3) The indexing parameters m_1 and m_2, if variable, may assume positive, negative or zero values.

4) i is initially m_1. As soon as i exceeds m_2, the loop is terminated.

5) DO loops may be nested 50 deep.

6) The value of a replacement statement outside or within a DO-loop should not exceed $2^{15}-1$ if the replacement variable is the index variable for the DO-loop and a second or third variable subscript in a double or triple dimension array.

```
         .
         .
         .
         .
         .
   J = 2525252525252526B
         .
         .
         .
         .
         .
   DO 2  J = 1, 3
   DO 2  I = 1, 3
         .
         .
         .
         .
         .
3  IARRAY (I,J) = 1
         .
         .
         .
```

The indexing of IARRAY is miscalculated since J was previously assigned a value exceeding $2^{15}-1$.

6.6

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.

6.7

PAUSE

PAUSE

PAUSE n

n is an octal number without a B suffix. 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 with the statement immediately following PAUSE. Although n is octal, a B suffix will cause a diagnostic.

6.8

STOP

STOP

STOP n

n is an octal number without a B suffix. 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. STOP (n omitted) causes immediate exit to monitor. A B suffix will cause a diagnostic if used with n.

6.9

END

END

END marks the physical end of a program or subprogram. It is executable in the sense that it will effect return from a subprogram in the absence of a RETURN. When used in a subprogram where it is immediately preceded by a transfer statement such as RETURN, GO TO, it marks the physical end of the subprogram.

The END statement may include the name of the program or subprogram which it terminates. This name, however, is ignored.

FUNCTIONS AND SUBPROGRAMS 7

Sets of instructions may be written as independent subroutines or function subprograms which can be referred to by the main program. The mode of a function subprogram is determined by the name of the subroutine in the same manner as variable modes are determined. A function subprogram must have at least one parameter and may have as many as 63; it returns a single value.

Subroutine subprogram names are not classified by mode. They may have, none or from one to 63 parameters and may return one value, several values, or no value. The name of a function or subroutine must be unique within that subroutine or function.

7.1

MAIN PROGRAMS AND SUBPROGRAMS

A main program may be written with or without references to subprograms. In all cases, the first statement must be of the following form where name is an alphanumeric identifier, 1-8 characters. The first character must be alphabetic; the remaining characters may be alphabetic or numeric.

PROGRAM name

A main program may refer to both subroutines and functions which are compiled independently of the main program. A calling program is a main program or subprogram that refers to subroutines and functions.

In a PROGRAM statement, if the name is followed by parameters, the program is treated as a subroutine except the name will become the transfer name on the transfer (TRA) card.

PROGRAM name $(p_1, p_2, \ldots p_n)$

This statement is used to pass parameters to overlays and segments. (COOP Monitor/Programmer's Guide, publication No. 530a.)

7.2

FUNCTION SUBPROGRAM

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

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

The name of a function subprogram may occur as an operand in an arithmetic statement. The function reference must supply the function with at least one argument and it may contain up to 63. The form of the function reference is:

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

F is the function name and 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. Because formal parameters are local to the subprogram in which they appear, they may be the same as variable names appearing in another subprogram.

The first statement of function subprograms must have the form:

$$\text{FUNCTION}\ \ F\ (p_1, p_2, \ldots p_n) \quad 1 \leq n \leq 63$$

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.

Rules:

1 The type of the function is determined from the naming conventions specified for variables in Chapter 4. (TYPE Declarations.)

2 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

3 No element of a formal parameter list may appear in a COMMON, EQUIVALENCE, DATA, OR EXTERNAL statement within the function subprogram. If it does, a compiler diagnostic results.

4 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.

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

arithmetic expression

constant or variable, simple or subscripted

array name

function reference

subroutine

When the name of a function subprogram appears as an actual parameter, that name must also appear in an EXTERNAL statement in the calling program. Since a function must always return a single value, it may appear as one parameter or two parameters:

1) two parameters

```
FUNCTION PULL (X,Y)
          .
          .
          .
B=X(Y)
          .
          .
          .
```

Function Subprogram Reference

```
          .
          .
          .
A=PULL (SINF,X)
          .
          .
          .
```

2) one parameter

```
FUNCTION PULL (X)
          .
          .
          .
B=X
          .
          .
          .
```

Function Subprogram Reference

```
          .
          .
A=PULL (SINF(X))
          .
          .
```

When a subroutine appears as an actual parameter, the subroutine name may appear alone or with a parameter list. When a subroutine appears with a parameter list, the subroutine name and its parameters must appear as separate actual parameters:

```
FUNCTION PULL (X,Y,Z)
        .
        .
        .
CALL X(Y,Z)
        .
        .
        .
```

<u>Subroutine Subprogram Reference</u>

```
        .
        .
A=PULL(DIS,A,B)
        .
        .
```

6 Logical expressions may not be actual parameters.

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

8 Functions must have at least one parameter.

7.3 LIBRARY FUNCTIONS

Function subprograms that are used frequently have been written and stored in a reference library and are available to the programmer through the compiler.

FORTRAN-63 contains the standard library functions available in earlier versions of FORTRAN. A list of these functions is in Appendix C. When one appears in the source program, the compiler identifies it as a library function and generates a special calling sequence within the object program.

In the absence of a TYPE declaration, the type of the function identifier is determined by its first letter. However, for standard library functions the modes of the results have been established through usage. The compiler recognizes the standard library functions and associates the established types with the results.

For example, in the function identifier LOGF, the first letter, L, would normally cause that function to return an integer result. This is contrary to established FORTRAN usage. The compiler recognizes LOGF as a standard library function and permits the return of a real result.

7.4
EXTERNAL STATEMENT

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

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

Identifier i 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; it should not be used with arithmetic statement functions. If it is, a compiler diagnostic is given.

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
END
```

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.

7.5
STATEMENT FUNCTIONS

Statement functions are defined when used as an operand in 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

$$F\ (p_1,p_2, . . .p_n) = E \quad 1 \le n \le 63$$

F is the function name, p_i are the actual parameters, and E is an expression.

Rules:

1 The type of the function is determined from the naming conventions specified for variables in Chapter 4, TYPE Declarations.

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

3 The formal parameters will usually appear in the expression E. When the statement function is executed, 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. 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. Formal parameters must be simple variables.

4 E may be arithmetic or logical.

5 E may contain subscripted variables, but the subscripts are restricted to integer constants.

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

7 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 $Z(x,y)=e^{x+iy}$.

7.6 SUBROUTINE SUBPROGRAM

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

The first statement of subroutine subprograms must have the form:

SUBROUTINE S

or

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

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

Rules:

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

2 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.

3 No element of a formal parameter list may appear in a COMMON, EQUIVALENCE, DATA, or EXTERNAL statement within the subroutine subprogram.

4 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.

7.7 CALL

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

CALL S

or

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

S is the subroutine name, and p_i are the actual parameters. The CALL statement transfers control to the subroutine. When a RETURN or END statement is encountered in the subroutine, control is returned to the next executable statement following the CALL in the calling program. If the CALL statement is the last statement in a DO, looping continues until satisfied. 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.

Rules:

1 The subroutine returns values through formal parameters which are substituted for actual parameters or through common variables. No value is associated with its name.

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

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

   arithmetic expression

   constant or variable, simple or subscripted

   array name

   function reference

   subroutine or function name

   When the name of a function subprogram appears as an actual parameter, that name must also appear in an EXTERNAL statement in the calling program. Since a function must always return a single value, it may appear as one or two parameters.

   1) two parameters

```
FUNCTION PULL (X,Y)
          .
          .
          .
B = X (Y)
          .
          .
          .
```

   Function Subprogram Reference

```
          .
          .
          .
A = PULL (SINF,X)
          .
          .
          .
```

   2) one parameter

```
FUNCTION PULL (X)
          .
          .
          .
B = X
          .
          .
          .
```

```
Function Subprogram Reference
          .
          .
          .
A = PULL (SINF (X))
          .
          .
          .
```

When a subroutine appears as an actual parameter, the subroutine name may appear alone or with a parameter list.

When a subroutine appears with a parameter list, the subroutine name and its parameters must appear as separate actual parameters.

```
FUNCTION PULL (X,Y,Z)
          .
          .
          .
CALL X(Y,Z)
          .
          .
          .
Subroutine Subprogram Reference
          .
          .
A = PULL (DIS,A,B)
          .
          .
```

4 Because formal parameters are local to the subroutine in which they appear, they may be the same as names appearing outside the subroutine.

5 Actual and formal parameters must agree in order number and type.

6 Logical expressions may not be actual parameters.

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)
```

7.8

PROGRAM ARRANGEMENT

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 main program and subprograms follows.

```
PROGRAM SOMTHING
     .
     .
     .
END

SUBROUTINE S1
     .
     .
     .
END

SUBROUTINE S2
     .
     .
     .
END
     .
     .
     .
FUNCTION F1 (. . .)
     .
     .
     .
END

FUNCTION F2 (. . .)
     .
     .
     .
END
```

7.9

RETURN AND END

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. The form is:

```
RETURN
```

In function references, control returns to the statement containing the function. In subroutine subprograms, control, in most cases, returns to the calling program.

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.

A RETURN statement in the main program causes an exit to the monitor.

7.10
ENTRY

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

```
ENTRY name
```

Name is an alphanumeric identifier, and may appear within the subprogram only in the ENTRY statement. Each entry identifier must appear in a separate ENTRY statement. The maximum number of entry points, including the subprogram name, is 20. The formal parameters, if any, appearing with the FUNCTION or SUBROUTINE statement do not appear with the ENTRY statement. ENTRY may appear anywhere within the subprogram except it should not appear within a DO; it cannot be labeled.

In the calling program, the reference to the entry name is made just as if reference were being made to the FUNCTION or SUBROUTINE in which the ENTRY is imbedded. Rules 5 and 6 of 7.2 apply.

ENTRY names must agree in type with the function or subroutine name.

Examples:

```
    FUNCTION JOE(X,Y)
10  JOE=X+Y
    RETURN
    ENTRY JAM
    IF(X.GT.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+JAM(Q,2.*P)
```

7.11

VARIABLE DIMENSIONS IN 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 subprogram. In the subroutine call from the calling program, the corresponding actual parameters specified are used by the called subprogram. The maximum dimension that any given array may assume is determined by a DIMENSION statement in the main program at compile time.

Rules:

1 The rules of 7.2, 7.5, and 7.7 apply

2 The formal parameters representing the array dimensions must be simple integer variables. The array identifier must also be a formal parameter.

3 The actual parameters representing the array dimensions may be integer constants or integer variables.

4 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 may not exceed N.

Examples:

1) Consider a simple matrix add routine written as a subroutine:

```
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(10,5), C(10,4), D(10,2)
```

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,10,4)
                   .
                   .
                   .
    CALL MATADD (A,B,D,10,2)
                   .
                   .
                   .
    CALL MATADD (B,C,D,10,2)
```

As the dimensions of a given array are changed, the reference point of any specific element may also be changed. For example:

```
    PROGRAM      MAD
    DIMENSION    A(6,7) C(5,5)
    DO 1  J=1,5
    DO 1  I=1,5
1   A(I,J) =  I+J
    CALL  MADX(C,A,5,5)
      ↓
    END

    SUBROUTINE  MADX (X,Y,M,N)
    DIMENSION    X(M,N), Y(M,N)
    DO  10  I = 1,N
    DO  10  J = 1,M
10  X(I,J) =  Y(I,J)
    END
```

A(I,J) references elements in an array defined as 6 x 7. Whereas Y(I,J) is referencing elements according to an array defined to be 5 x 5 in this particular calling statement.

2)

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

Its transpose Y^1 is:

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

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

FORMAT SPECIFICATIONS 8

Data transmission between storage and an external unit requires the FORMAT statement and the I/O control statement (Chapter 9). The I/O statement specifies the input/output device and process--READ, WRITE, and so forth, and a list of data to be moved. The FORMAT statement specifies the manner in which the data is to be moved. In binary tape statements no FORMAT statement is used.

8.1
THE I/O LIST

The list portion of an I/O control statement indicates the data elements and the order, from left to right, of transmission. Elements may be simple variables, array names (subscripted or non-subscripted), or constants on output only. List elements are separated by commas, and the order must correspond to the order of the FORMAT specifications.

Subscripts in an I/O list may be one of the following forms:

```
(c*I±d)
(I±d)
(c*I)
(I)
(c)
```

c and d are unsigned integer constants: and I is a simple integer variable, previously defined, or defined within an implied DO loop.

Examples:

```
A,B,H(I),Q(3,4)
SPECS
A,DELTAX(J+1)
```

8.1.1
TRANSMISSION OF ARRAYS

Part or all of an array can be represented as a list item. Multi-dimensioned arrays may appear in the list, with values specified for the range of the subscripts in an implied DO loop.

The general form is:

$(((A(I,J,K),\ \gamma_1=m_1,m_2,m_3),\ \gamma_2=n_1,n_2,n_3),\ \gamma_3=p_1,p_2,p_3)$

m_i,n_i,p_i	are unsigned constants or predefined positive integer variables. If m_3,n_3 or p_3 is omitted it is construed as 1.
I,J,K	are subscripts of A and must be of the standard form.
$\gamma_1,\gamma_2,\gamma_3$	are I, J, or K: $\gamma_1 \neq \gamma_2 \neq \gamma_3$

The I/O list may contain nested implied DO loops to a maximum depth of 50.

Example:

DO loops nested 5 deep:

$(((((A(I,J,K),B(M),\ C(N),\ N=n_1,n_2,n_3),\ M=m_1,m_2,m_3),\ K=k_1,k_2,k_3),$
$J=j_1,j_2,j_3),\ I=i_1,i_2,i_3)$

During execution, each subscript (index variable) is set to the initial index value: $I=i_1$, $J=j_1$, $K=k_1$, $M=m_1$, $N=n_1$.

The first index variable defined in the list is incremented first. Data named in the implied DO loops is transmitted in increments according to the third DO loop parameter until the second DO loop parameter is exceeded. If the third parameter is omitted, the increment value is 1. When the first index variable reaches the maximum value, it is reset; the next index variable to the right is incremented and the process is repeated until the last index variable has been incremented.

An implied DO loop may also be used to transmit a simple variable more than one time. In (A,K=1,10), A will be transmitted 10 times.

Example:

As an element in an input/output list, the expression

$(((A(I,J,K),I=m_1,m_2,m_3),\ J=n_1,n_2,n_3),\ K=p_1,p_2,p_3)$

implies a nest of DO loops of the form

DO 10 $K=p_1,p_2,p_3$
DO 10 $J=n_1,n_2,n_3$
DO 10 $I=m_1,m_2,m_3$
Transmit A(I,J,K)
10 CONTINUE

To transmit the elements of a 3 by 3 matrix by columns:

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

To transmit the elements of a 3 by 3 matrix by rows:

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

If a multi-dimensioned array name appears in a list without subscripts, the entire array is transmitted.

For example, a multi-dimension non-subscripted list element, SPECS, with an associated DIMENSION SPECS (7,5,3) statement is transmitted as if under control of the nested DO loops.

```
   DO    10    K=1,3
   DO    10    J =1,5
   DO    10    I =1,7
   Transmit SPECS(I,J,K)
10 CONTINUE
```

or as if under control of an implied DO loop,

```
. . . ,( (SPECS(I,J,K), I=1,7), J=1,5), K=1,3), . . .
```

I/O Lists:

```
( (BUZ(K,2*L),K=1,5), L=1, 13,2)
Q(3), Z(2,2), (TUP(3*I-4), I=2,10)
(RAZ(K), K=1, LIM1, LIM2)
```

8.2

FORMAT STATEMENT

The BCD I/O control statements require a FORMAT statement which contains the specifications relating to the internal-external structure of the corresponding I/O list elements.

$$\text{FORMAT } (\text{spec}_1, \ldots, k(\text{spec}_m, \ldots), \text{spec}_n, \ldots)$$

Spec_i is a format specification and k is an optional repetition factor which must be an unsigned integer constant. The FORMAT statement is non-executable, and may appear anywhere in the program.

8.3
FORMAT SPECIFICATIONS

The data elements in I/O lists are converted from external to internal or from internal to external representation according to FORMAT conversion specifications. FORMAT specifications also may contain editing codes.

FORTRAN-63 conversion specifications

Ew.d	Single precision floating point with exponent
Fw.d	Single precision floating point without exponent
Dw.d	Double precision floating point with exponent
C(Zw.d,Zw.d)	Complex conversion; Z may be E or F conversion
Iw	Decimal integer conversion
Ow	Octal integer conversion
Aw	Alphanumeric conversion
Rw	Alphanumeric conversion
Lw	Logical conversion
nP	Scaling factor

FORTRAN-63 editing specifications

wX	Intra-line spacing
wH	Heading and labeling
/	Begin new record

Both w and d are unsigned integers. w specifies the field width, the number of character positions in the record; and d specifies the number of digits to the right of the decimal within the field.

8.4
CONVERSION SPECIFICATIONS

8.4.1
Ew.d OUTPUT

E conversion is used to convert floating point numbers in storage to the BCD character form for output. The field occupies w positions in the output record; the corresponding floating point number will appear right justified in the field as

$\pm\alpha.\alpha. \ldots .\alpha$	E-ee	$1 \leq ee \leq 99$
$\pm\alpha.\alpha. \ldots .\alpha$	E ee	$0 \leq ee \leq 99$
$\pm\alpha.\alpha. \ldots .\alpha$	Eeee	$100 \leq ee \leq 307$
$\pm\alpha.\alpha. \ldots .\alpha$	-eee	

$\alpha.\alpha. \ldots .\alpha$ are the most significant digits of the integer and fractional part and ee and eee are the digits in the exponent. If d is zero or blank, the decimal point and digits to the right of the decimal do not appear as shown above. Positive signs are suppressed and the exponent signs appear as shown above. The fractional part contains a maximum of 11 digits. Field w must be wide enough to contain the significant digits, signs, decimal point, E, and the exponent.

If the field is not wide enough to contain the output value, digits are dropped from the right of the fraction and the fraction sign may be suppressed. An asterisk is inserted immediately before the designator E if a negative sign or digits or both are lost. A field width, w, less than five will give a format error. If the field is longer than the output value, the quantity is right justified with blanks in the excess positions to the left.

For P-scaling on output, see section 8.6.2.

Examples:

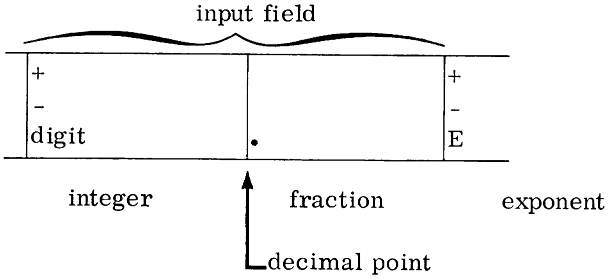
Ew.d Output

```
    PRINT 10, A                          A contains -67.32
10  FORMAT(E10.3)                                or +67.32

    Result:  -6.732E∧01 or ∧6.732E∧01

    PRINT 10, A
10  FORMAT(E13.3)

    Result:    ∧∧∧-6.732E∧01 or   ∧∧∧∧6.732E∧01

    PRINT 10, A                          A contains -67.32
10  FORMAT(E9.3)

    Result:  6.73*E∧01
                                         provision not made for sign
    PRINT 10, A
10  FORMAT(E10.4)

    Result:  6.732*E∧01
```

8.4.2
Ew.d INPUT

The E specification converts the number in the input field (specified by w) to a real and stores it in the appropriate location in memory.

Subfield structure of the input field:



The total number of characters in the input field is specified by w; this field is scanned from left to right.

An integer subfield begins with a sign (+ or -) or a digit and may contain a string of digits (a sequence of consecutive numbers); blanks are interpreted as zeros. The integer field is terminated by a decimal point, an E, a + or -, or the end of the input field.

A fraction subfield which begins with a decimal point may contain a string of digits. The field is terminated by an E, a + or -, or the end of the input field.

An exponent subfield may begin with an E, a + or -. When it begins with E, the + or - may appear between E and the string of digits of the subfield. The value of a string of digits in this subfield must be less than 310.

Permissible subfield combinations:

+1.6327E-04	integer fraction exponent
-32.7216	integer fraction
+328+5	integer exponent
.629E-1	fraction exponent
+136	integer only
.07628431	fraction only
E-06 (interpreted as zero)	exponent only

Rules:

1. In the Ew.d specification, d acts as a negative power of ten scaling factor when the fraction subfield is not present. The internal representation of the input quantity will be:

   $$\text{(integer subfield) } x10^{-d} x10^{\text{(exponent subfield)}}$$

   For example, if the specification is E7.8, the input quantity 3267+05 will be converted and stored as: $3267 \; x10^{-8} x10^{5} = 3.267$.

2. If E conversion is specified, but a decimal point occurs in the input constants, the decimal point will override d. The input quantity 3.67294+5 may be read by any specification but will always be stored as $3.67294x10^{5}$.

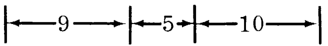
3. When d does not appear it is assumed to be zero.

4. The maximum number of significant digits that may appear in the combined integer-fraction field is 11. Excess digits to the right are lost during the conversion process.

5. The field length specified by w in Ew.d should always be the same as the length of the input field containing the input number. When it is not, incorrect numbers may be read, converted and stored as shown below. The field w includes the significant digits, signs, decimal point, E, and exponent.

```
      READ 20,A,B,C
20    FORMAT (E9.3,E7.2,E10.3)
```

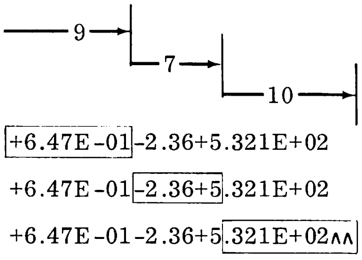
The input quantities appear on a card in three contiguous field columns 1 through 24:



+6.47E-01-2.36+5.321E+02

The second specification (E7.2) exceeds the physical field width of the second value by two characters.

Reading proceeds as follows:



+6.47E-01-2.36+5.321E+02

+6.47E-01-2.36+5.321E+02

+6.47E-01-2.36+5.321E+02∧∧

First +6.47-01 is read, converted and placed in location A.

Next, -2.36+5 is read, converted and placed in location B. The number actually desired was -2.36, but the specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input representation under the definitions and restrictions.

Finally .321E+02∧∧ is read, converted and placed in location C. Here again, the input number is legitimate; and it is converted and stored, even though it is not the number desired.

The above example illustrates a situation where numbers are incorrectly read, converted, and stored, and yet there is no immediate indication that an error has occurred.

Examples:

Ew.d Input

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	.14326	All subfields present
-12.437629E+1	E13.6	-124.37629	All subfields present
8936E+004	E9.10	.008936	No fraction subfield. Input number converted as 8936. x 10^{-10+4}
327.625	E7.3	327.625	No exponent subfield
4.376	E5	4.376	No d in specification
-.0003627+5	E11.7	-36.27	Integer subfield contains - only
-.0003627E5	E11.7	-36.27	Integer subfield contains - only
blanks	Ew.d	-0.	All subfields empty
1E1	E3.0	10.	No fraction subfield. Input number converted as 1. x10^{1}
E+06	E10.6	0.	No integer or fraction subfield. Zero stored regardless of exponent field contents.

8.4.3

Fw.d OUTPUT

The field occupies w positions in the output record; the corresponding list element must be a floating point quantity, and it will appear as a decimal number, right justified in the field w, as:

$$\pm \delta \ldots \delta . \delta \ldots \delta$$

δ represents the most significant digits of the number (maximum 11). The number of decimal places to the right of the decimal is specified by d. If d is zero or omitted, the decimal point and digits to the right do not appear. If the number is positive, the + sign is suppressed.

If the field is too short to accommodate the number, characters are discarded from the right, the fraction sign is suppressed and an asterisk appears in the last character position to indicate the error.

If the field w is longer than required to accommodate the number, it is right justified with blanks occupying the excess field positions to the left.

If the magnitude of the internal number representation after P-scaling exceeds $2^{47}-1$, F conversion outputs a blank field.

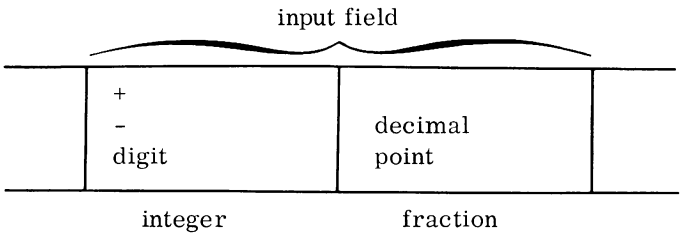
Examples:

```
A contains +32.694

     PRINT 10,A
10   FORMAT(F7.3)

     Result: ∧ 32.694

     PRINT 11,A
11   FORMAT(F10.3)

     Result:∧∧∧∧ 32.694

A contains -32.694

     PRINT 12,A
12   FORMAT(F6.3)          (
                            no provision for - sign
     Result:  32.69*       (
```

8.4.4 Fw.d INPUT

This specification is a modification of Ew.d. The input field consists of an integer and a fraction subfield. An omitted subfield is assumed to be zero.

Subfield structure of the input field:



An integer subfield begins with a digit, + or -; it may contain a string of digits, (a sequence of consecutive numbers). Blanks in the string are interpreted as zeros. The integer field is terminated by a period, or by the end of the input field.

A fraction subfield begins with a decimal point and may contain a string of digits; it is terminated by the end of the input field.

The following subfield combinations are permissible:

Integer fraction	-32.7216
Integer by itself	+1326
Fraction by itself	.719325684

Rules:

1 In the Fw.d specification, d acts as a negative power of ten scaling factor when the fraction subfield is not present. The internal representation is: (integer subfield) x 10^{-d}. For example, the specification F4.4 causes the input quantity 3267 to be converted and stored as 3267 x 10^{-4} = .3267.

2 A decimal point in the input quantity causes d to be ignored. For example, 3.6789 may be read under any specifications but will always be stored as 3.6789.

3 When d does not appear it is assumed to be zero. For example, the input quantity +14.62 is read into memory by the specification F6 as 14.62.

4 The maximum number of significant digits that may appear in the combined integer-fraction field is 11. Excess digits are discarded during the conversion process from the right.

5 The field length specified by w in Fw.d should always be the same as the actual length of the input field containing the input number. When it is not, incorrect numbers may be read, converted and stored. See example under rule 5, section 8.4.2.

Examples:

Fw.d Input

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field
37925	F5.7	.0037925	No fraction subfield. Input number converted as 37925 x 10^{-7}
-4.7366	F7	-4.7366	No d in specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.d	.62543	Decimal point overrides d of specification.
+144.15E-03	F11.2	.14415	Exponents are legitimate in F input and may have P-scaling.

8.4.5
Dw.d OUTPUT

The field occupies w positions of the output record, the corresponding list element which must be a double precision quantity will appear as a decimal number, right justified in the field w as:

$\pm\alpha.\alpha\ldots\ldots\alpha$	E-ee	$1 \leq ee \leq 99$
$\pm\alpha.\alpha\ldots\ldots\alpha$	E ee	$0 \leq ee \leq 99$
$\pm\alpha.\alpha\ldots\ldots\alpha$	Eeee	$100 \leq eee \leq 307$
$\pm\alpha.\alpha\ldots\ldots\alpha$	-eee	

D conversion corresponds to Ew.d Output except that 25 is the maximum number of digits in the fraction. P-scaling is not applicable.

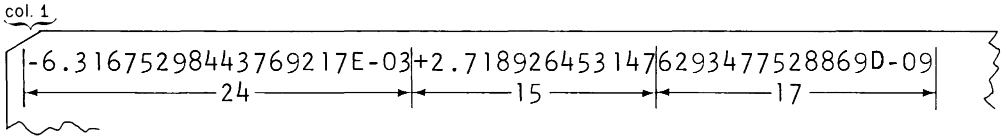
8.4.6
Dw.d INPUT

D conversion corresponds to Ew.d Input except that 25 is the maximum number of significant digits permitted in the combined integer-fraction field. P-scaling is not applicable. D is acceptable in place of E as the beginning of an exponent field.

Example:

```
     TYPE DOUBLE Z,Y,X
     READ1,  Z,Y,X
1    FORMAT (D24.17,D15,D17.4)
```

Input card:



8.4.7
$C(Z_1w_1.d_1, Z_2w_2.d_2)$ OUTPUT

Z is either E or F. The field occupies $w_1 + w_2$ positions in the output record, and the corresponding list element must be complex. $w_1 + w_2$ are two real values; w_1 represents the real part of the complex number and w_2 represents the imaginary part. The value may be one of the following forms:

$\pm\,\delta\,.\,\delta\,\ldots\,\delta$ Exp. $\pm\,\delta\,.\,\delta\,\ldots\,\delta$ Exp.	(Ew.d,Ew.d)
$\pm\,\delta\,.\,\delta\,\ldots\,\delta$ Exp. $\pm\,\delta\,\ldots\,\delta\,.\,\delta\,\ldots\,\delta$	(Ew.d,Fw.d)
$\pm\,\delta\,\ldots\,\delta\,.\,\delta\,\ldots\,\delta \pm \delta\,.\,\delta\,\ldots\,\delta$ Exp.	(Fw.d,Ew.d)
$\pm\,\delta\,\ldots\,\delta\,.\,\delta\,\ldots\,\delta \pm \delta\,\ldots\,\delta.\delta\,\ldots\,\delta$	(Fw.d,Fw.d)

Exp is:

$E\pm e_1 e_2$ if exponent ≤ 99

$E\ e_1 e_2 e_3$ if exponent > 99

$-e_1 e_2 e_3$ if exponent < -99

The restrictions for Ew.d and Fw.d apply.

If spaces are desired between the two output numbers, the second specification should indicate a field (w_2) larger than required.

Example:

```
    TYPE COMPLEX A
    PRINT 10,A
10  FORMAT (C(F7.2,F9.2) )
```

real part of A is 362.92

imaginary part of A is -46.73

Result: ∧ 362.92∧∧∧-46.73

8.4.8
C($Z_1w_1.d_1,Z_2w_2.d_2$) INPUT

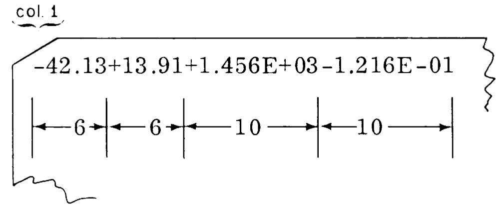
Z is either E or F and the input quantity occupies $w_1 + w_2$ character positions. The first w_1 characters are the representation of the real part of the complex number, and the remaining w_2 characters are the representation of the imaginary part of the complex number.

The restrictions for Ew.d and Fw.d apply.

Example:

```
    TYPE COMPLEX A,B
    READ 10,A,B
10  FORMAT (C(F6.2,F6.2),  C(E10.3,E10.3) )
```

Input card:



8.4.9
Iw OUTPUT

I specification is used to output decimal integer values. The output quantity occupies w output record positions; it will appear right justified in the field w, as:

$$\pm \delta \ . \ . \ . \ \delta$$

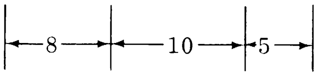
δ is the most significant decimal digits (maximum 15) of the integer. If the integer is positive the + sign is suppressed.

If the field w is larger than required, the output quantity is right justified with blanks occupying excess positions to the left. If the field is too short, characters are discarded from the left and an asterisk appears in the last field position.

Example:

```
    PRINT 10,I,J,K                        I contains -3762
10  FORMAT (I8,I10,I5)                    J contains +4762937
                                          K contains +13
```

Result: ∧∧∧-3762∧∧∧4762937∧∧∧13



8.4.10
Iw INPUT

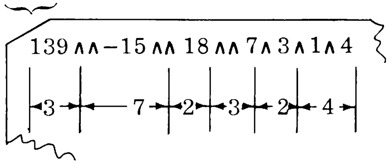
The field is w characters in length and the corresponding list element must be a decimal integer quantity.

The input field w which consists of an integer subfield may contain only the characters +, -, the digits 0 through 9, or blank. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. The value is stored right-justified in the specified variable.

Example:

```
    READ 10,I,J,K,L,M,N
10  FORMAT (I3,I7.I2,I3.I2,I4)
```

Input card:



In memory:

I contains	139
J	-1500
K	18
L	7
M	3
N	104

8.4.11

Ow OUTPUT

O specification is used to output octal integer values. The output quantity occupies w output record positions, and it will appear right justified in the field as: δ δ . . . δ

δ are octal digits, and leading zeros are suppressed. If w is 16 or less, the rightmost w digits appear. If w is greater than 16, the number is right justified in the field with blanks to the left of the output quantity. A negative number is output in its complement form.

8.4.12

Ow INPUT

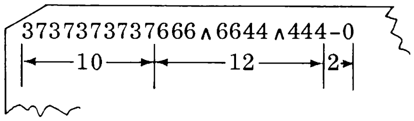
Octal integer values are converted under O specification. The field is w octal integer characters in length and the corresponding list element must be an integer quantity.

The input field w consists of an integer subfield only (maximum of 16 octal digits). The only characters that may appear in the field are +, or -, blank and 0 through 7. Only one sign is permitted; it must precede the first digit in the field. Blanks are interpreted as zeros.

Example:

```
      TYPE INTEGER P,Q,R
      READ 10,P,Q,R
10    FORMAT (O10,O12,O2)
```

Input Card:



```
In memory:   P:   0000003737373737

             Q:   0000666066440444

             R:   7777777777777777  A negative number is
                                    represented in complement
                                    form.
```

A negative octal number is represented internally in 16-digit seven's complement form obtained by subtracting each digit of an octal number from seven. For example, if -703 is an input quantity, its internal representation is 7777777777777074. That is,

```
  7777777777777777
- 0000000000000703
  ----------------
  7777777777777074
```

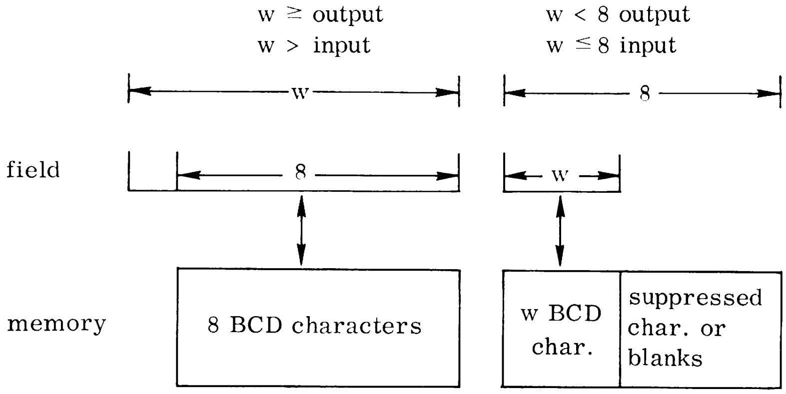
8.4.13
Aw OUTPUT

A conversion is used to output alphanumeric characters. If w is 8 or more, the output quantity appears right justified in the output field, blank fill to left. If w is less than 8, the output quantity represents the leftmost w characters, left justified in the field.

8.4.14
Aw INPUT

This specification will accept as list elements any set of six bit characters including blanks. The internal representation is BCD; the field width is w characters.

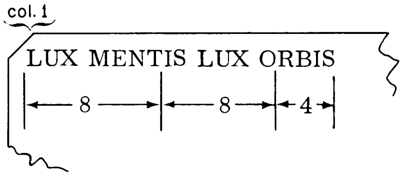
If w exceeds 8, the input quantity will be the rightmost 8 characters. If w is 8 or less, the input quantity goes to the designated storage location as a left justified BCD word, the remaining spaces are blank-filled.



Example: (Compare with next example)

```
      READ 10,Q,P,O
   10 FORMAT (A8,A8,A4)
```

Input card:



In memory: Q: LUXbMENT

P: ISbLUXbO

O: RBISbbbb

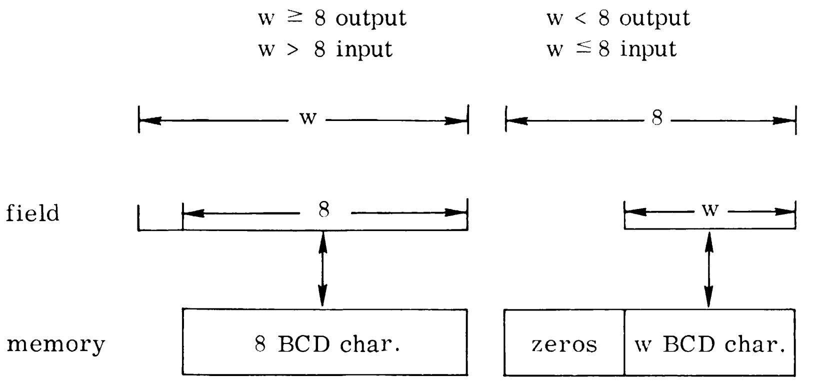
8.4.15
Rw OUTPUT

This specification is the same as the Aw specification with the following exception.

If w is less than 8, the output quantity represents the rightmost characters.

8.4.16
Rw INPUT

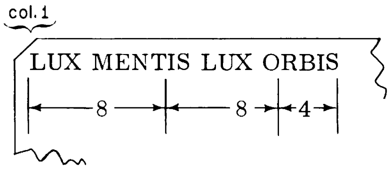
If w is less than 8, the input quantity goes to the designated storage location as a right justified binary zero filled word.



Example: (Compare with previous example)

```
    READ 10,Q,P,O
10  FORMAT (R8,R8,R4)
```

Input card:



In memory:

Q: LUXbMENT

P: ISbLUXbO

O: 0000RBIS

8.4.17
Lw OUTPUT

L specification is used to output logical values. The input/output field is w characters long and the corresponding list element must be a logical element

If w is greater than 1, 1 or 0 is printed right justified in the field w with blank fill to the left.

Example:

```
    TYPE LOGICAL I,J,K,L
    PRINT 5,I,J,K,L
5   FORMAT (4L3)
```

I contains 1
J contains 0
K contains 1
L contains 1

Result: ∧∧1∧∧0∧∧1∧∧1

8.4.18
Lw INPUT

This specification will accept logical quantities as list elements. A zero or a blank in the field w is stored as zero. A one in the field w is stored as one. Only one such character (0 or 1) may appear in any input field. Any character other than 0,1, or blank is incorrect.

8.5
EDITING SPECIFICATIONS

8.5.1
wX

This specification may be used to include w blanks in an output record or to skip w characters on input to permit spacing of input/output quantities.

Examples:

```
    PRINT 10,A,B,C                        A contains 7
10  FORMAT(I2,6X,F6.2,6X,E12.5)           B contains 13.6
                                          C contains 1462.37

    Result:   ∧7←— 6 —→∧13.60←— 6—→∧1.46237E+03

    READ 11,R,S,T
11  FORMAT(F5.2,3X, F5.2,6X,F5.2) or FORMAT (F5.2,3XF5.2,6XF5.2)
```

```
Input card:                         In memory:  R=14.62
  col.1                                         S=13.78
                                                T=15.97
 14.62∧∧$13.78 ∧ COST∧15.97
```

In the specification list, the comma following X is optional.

8.5.2
wH OUTPUT

This specification provides for the output of any set of six-bit characters, including blanks, in the form of comments, titles, and headings. w is an unsigned integer specifying the number of characters to the right of the H that will be transmitted to the output record. H denotes a Hollerith field. The comma following the H specification is optional.

Examples:

```
Source program:               PRINT 20
                          20  FORMAT(28H BLANKS COUNT IN AN H FIELD.)

produces output record:       ∧BLANKS COUNT IN AN H FIELD.

Source program:               PRINT 30,A                 A contains 1.5
                          30  FORMAT(6H LMAX=,F5.2) comma is optional

produces output record:       ∧LMAX=∧1.50
```

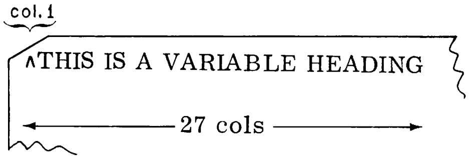
8.5.3
wH INPUT

The H field may be used to read a new heading into an existing H field.

Example:

```
Source program:          READ 10
                     10  FORMAT  (27H
```

Input card:



After READ the FORMAT statement labeled 10 will contain the alphanumeric information read from the input card; a subsequent reference to statement 10 in an output control statement would act as follows:

```
PRINT 10     produces the printer line:  ∧THIS IS A VARIABLE HEADING
```

8.5.4
NEW RECORD

The slash, /, which signals the end of a BCD record may occur anywhere in the specifications list. It need not be separated from the other list elements by commas; consecutive slashes may appear in a list. During output, it is used to skip lines, cards, or tape records. During input, it specifies that control passes to the next record or card. k lines will be skipped for (k(/)).

Examples:

```
    PRINT 10
10  FORMAT (20X,7HHEADING///6X,5HINPUT,19X,6HOUTPUT)
```

```
Print-out:     HEADING                                              line 1
                                                                    line 2
                                                                    line 3
    INPUT                OUTPUT                                     line 4
```

Each line corresponds to a BCD record. The second and third records are null and produce the line spacing illustrated.

```
    PRINT 11,A,B,C,D
11  FORMAT (2E10.2/2F7.3)
```

Internally:
A = -11.6
B = .325
C = 46.327
D = -14.261

```
    Result:      -1.16E 01 3.25E-01
                 46.327-14.261
```

```
    PRINT 11,A,B,C,D
11  FORMAT (2E10.2/ /2F7.3)

    Result:      -1.16E 01 3.25E-01

                 46.327-14.261
    PRINT 15, (A(I),I=1,9)
15  FORMAT (8H RESULTS2(/) (3F8.2) )
```

```
        RESULTS

           3.62    -4.03    -9.78
          -6.33     7.12     3.49
           6.21    -6.74    -1.18
```

8.6
nP SCALE FACTOR

A scale factor may precede the F conversion and E conversion. The scale factor is: External number = Internal number x10$^{\text{scale factor}}$. The scale factor applies to Fw.d on both input and output and to Ew.d on output only. A scaled specification is written in FORTRAN-63 as:

$$nP \begin{Bmatrix} E \\ F \end{Bmatrix} w.d$$

n is a signed integer constant which cannot exceed 13 for output. The nP specification may appear with complex conversion, C(Zw.d,Zw.d); each word is scaled separately according to Fw.d or Ew.d scaling.

8.6.1
Fw.d SCALING

Input

The number in the input field is divided by 10^n and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is 314.1592X10^{-2} = 3.141592.

Output

The number in the output field is the internal number multiplied by 10^n. In the output representation, the decimal point is fixed; the number moves to the left or right depending on whether the scale factor is plus or minus. For example, the internal number 3.1415926536 may be represented on output under scaled F specifications as follows:

Specification	Output Representation
F13.6	3.141593
1PF13.6	31.415927
3PF13.6	3141.592654
-1PF13.6	.314159

8.6.2
Ew.d SCALING

Output

The scale factor has the effect of shifting the output number left n places while reducing the exponent by n. Only positive n is permitted. Using 3.1415926538 some output representations corresponding to scaled E-specifications are:

Specification	Output Representation
E20.2	3.14E 00
1PE20.2	31.42E-01
2PE20.2	314.16E-02
3PE20.2	3141.59E-03
4PE20.2	31415.93E-04
5PE20.2	314159.27E-05

8.6.3
SCALING RESTRICTIONS

The scale factor is assumed to be zero if no other value has been given; however, once a value has been given, it will hold for all E and F specifications following the scale factor within the same FORMAT statement. To nullify this effect in subsequent E and F specifications, a zero scale factor, 0P, must precede an E or F specification. Scale factors for E and F output specifications must be in the range $-13 \leq n \leq 13$.

Scale factors on E input specifications are ignored.

The scaling specification nP may appear independently of an E or F specification, but it will hold for all E and F specifications that follow within the same FORMAT statement unless changed by another nP.

```
(3P, 3I9, F10.2)  same as
(3I9, 3PF10.2)
```

8.7

REPEATED FORMAT SPECIFICATIONS

Any FORMAT specification may be repeated by using an unsigned integer constant repetition factor, k, as follows: k(spec), spec is any conversion specification except nP.

For example, if two quantities K,L are to be printed, the program would be written:

```
      PRINT 10 K,L
  10  FORMAT (I2,I2)
```

Since the specifications for K,L are identical, the FORMAT statement may be written: 10 FORMAT (2I2)

When a group of FORMAT specifications repeats itself, as in FORMAT (E15.3,F6.1,I4,I4,E15.3,F6.1,I4,I4) the use of k produces: FORMAT (2(E15.3,F6.1,2I4))

In the above example, the parenthetical grouping of the FORMAT specifications is called a repeated group. A repeated group may not contain a repeated group: FORMAT (I6,2(F10.2,2I6,2E7.1)) is permitted, but FORMAT (I6,2(F10.2,2(I6, E7.1))) is not permitted.

8.7.1

UNLIMITED GROUPS

FORMAT specifications may be repeated without the use of a repetition factor. A parenthetical group that has no repetition factor is unlimited and will be used repeatedly until the I/O list is exhausted. Parentheses are the controlling factors in repetition. The right parenthesis of an unlimited group is equivalent to a slash. Specifications to the right of an unlimited group can never be reached.

The following are format specifications for output data:

```
      (E16.3,F20,7,(2I4,2(I3,F7.1) ),F8.2)
```

Print fields according to E16.3 and F20.7. Since 2(I3,F7.1) is a repeated parenthetical group, print fields according to (2I4,2(I3,F7.1)), which does not have repetition operator, until the list elements are exhausted. F8.2 will never be reached.

8.8

VARIABLE FORMAT

FORMAT lists may be specified at the time of execution. The specification list including left and right parentheses, but not the statement number or the word FORMAT, is read under A conversion or in a DATA statement and stored in an integer array. The name of the array containing the specifications may be used in place of the FORMAT statement number in the associated input/output operation. The array name that appears with or without subscript specifies the location of the first word of the FORMAT information.

Examples:

1) Assume the following FORMAT specifications:

```
(E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

This information could be punched in an input card and read by a program such as:

```
   DIMENSION IVAR(4)
   READ 1, (IVAR(I),I=1,4)
 1 FORMAT(3A8,A6)
```

The elements of the input card will be placed in storage as follows:

```
IVAR   :  (E12.2,F
IVAR+1 :  8.2,I7,2
IVAR+2 :  E20.3,F9
IVAR+3 :  .3,I4)bb
```

A subsequent output statement in the same program could refer to these FORMAT specifications as:

```
    PRINT IVAR(1),A,B,I,C,D,E,J
or
    PRINT IVAR,A,B,I,C,D,E,J
```

This would produce exactly the same result as the program:

```
    PRINT 10,A,B,I,C,D,E,J
10  FORMAT (E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

2)
```
DIMENSION LAIS(4)
DATA (LAIS=8H(E12.2,F8H8.2,2I7),8H(F8.2,E1,8H2.2,2I7) )
```

```
Output statements:          I = 1
                            PRINT LAIS(I),A,B,I,J
                         or
                            PRINT LAIS,A,B,I,J

which is the same as:       PRINT 1,A,B,I,J
                          1 FORMAT (E12.2,F8.2,2I7)

                            I = 3
                            PRINT LAIS(I),C,D,I,J

which is the same as:       PRINT 2,C,D,I,J
                          2 FORMAT (F8.2,E12.2,2I7)
```

INPUT/OUTPUT STATEMENTS

9

Input/output control statements transfer information between the storage unit and one of the following external devices:

- An 80 column card reader
- An 80 column card punch
- A 120 column printer
- A magnetic tape unit
- A typewriter

9.1

READ/WRITE STATEMENTS

The following definitions for i, n, L apply for all I/O control statements.

The logical unit number, i, must be an integer variable or a constant. Logical numbers are assigned to physical units by the monitor. The standard input unit is 50; standard output unit is 51; standard punch unit is 52.

The FORMAT statement describing the format of the data is represented by n which may be the statement number, a variable identifier or a formal parameter. Binary data transmission does not require a related FORMAT statement.

The input/output list is specified by L. Binary information is transmitted with odd parity checking bits. BCD information is transmitted with even parity checking bits.

9.1.1

WRITE STATEMENTS

PRINT n,L transfers information from the storage locations given by the list (L) to the standard output unit. This information is transferred as line printer images, 120 characters or less per line in accordance with the FORMAT statement, n. The maximum record length is 120 characters, but the first character of every record is used for carriage control[†] on the printer and is not printed.

[†] *

CHARACTER	ACTION
blank	single space after printing.
0	double space before printing.
1	eject page before printing.
+	suppress spacing after printing. Causes two successive records to be printed on the same line.

PUNCH n,L transfers information from the memory locations given by the list (L) identifiers to the standard punch unit. This information is transferred as card images, 80 characters or less per card in accordance with the FORMAT statement, n.

WRITE (i,n) L and **WRITE OUTPUT TAPE i,n,L**

are equivalent forms which transfer information from storage locations given by identifiers in the list (L) to a specified tape unit (i) according to the FORMAT statement (n). i may be 1 to 49 or 51, 52.

A logical record containing up to 120 characters is recorded on magnetic tape in even parity (BCD mode). Each logical record is one physical record. The number of words in the list (L) and the FORMAT statement (n) determine the number of records that will be written on a unit. If the logical record is less than 120 characters, the remainder of the record will be filled with blanks to the nearest multiple of 8 characters. All characters in excess of 120 will be lost and an error indication will be given.

The printer treats the first character of a record as a printer control character and does not print it. If the programmer fails to allow for a printer control character, the first character of the output data will be lost on the printed listing.

Examples:

```
      WRITE OUTPUT TAPE 10, 20, A, B, C
   20 FORMAT (3F10.6)

      TYPE DOUBLE D
      DIMENSION D (4)
      WRITE (10, 30) D
   30 FORMAT (4D25.16)

      WRITE OUTPUT TAPE 4, 21
   21 FORMAT (33H THIS STATEMENT HAS NO DATA LIST.)
```

WRITE (i) L and **WRITE TAPE i,L**

are equivalent forms which transfer information from storage locations given by the list (L) identifiers to a specified tape unit (i), i may be 1 to 49. If the list (L) is omitted, the WRITE (i) statement acts as a do-nothing statement.

The number of words in the list (L) determines the number of physical records that will be written on that unit. A physical record contains a maximum of 256 words — the first word is a control word, the remaining 255 contain the transmitted data. The last physical record may contain from 2 to 256 words. The physical records written by one WRITE (i) L statement constitutes one logical record. The information is recorded in odd parity (binary mode); the method is illustrated in figures 1a and 1b.

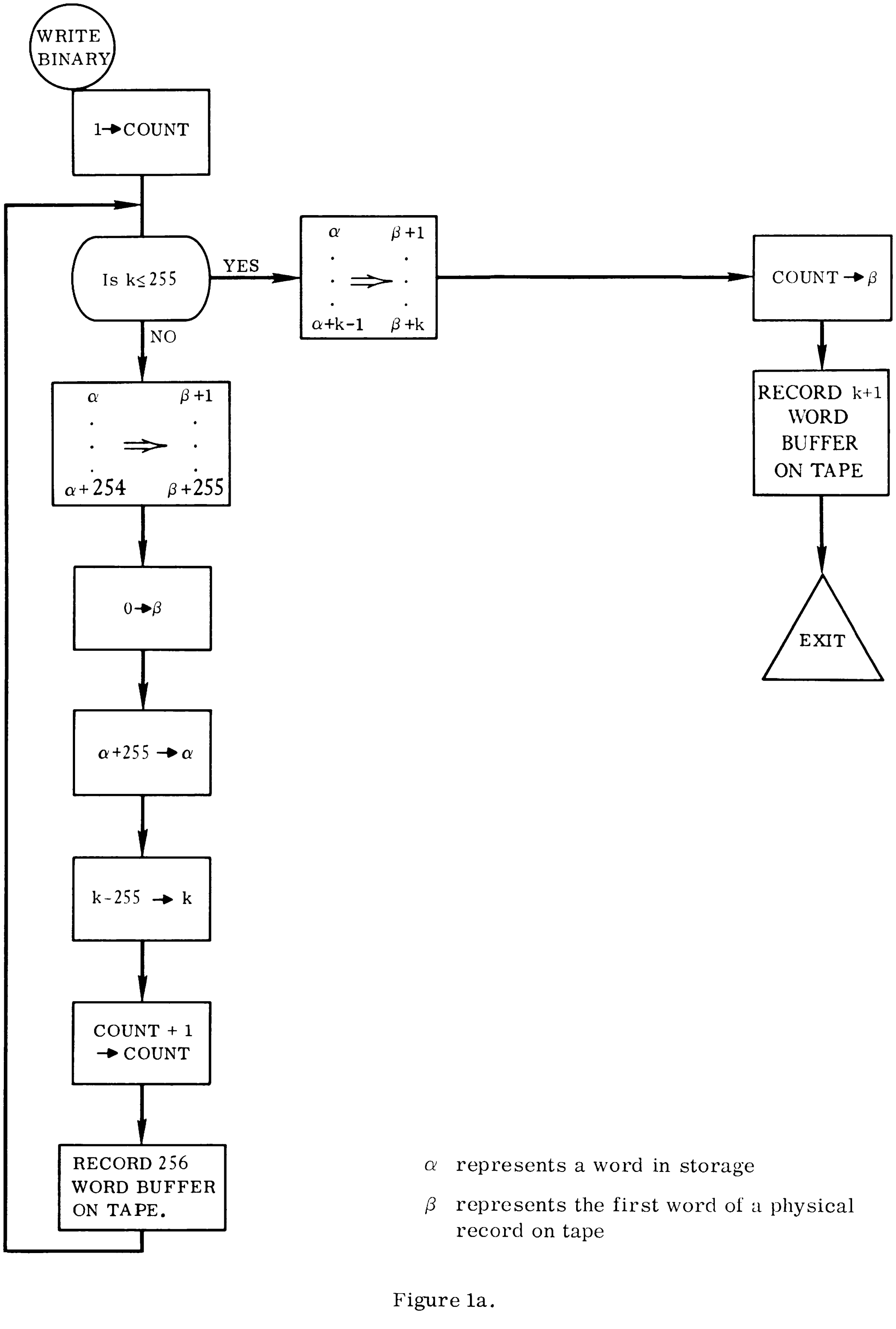
If there are n physical records in the logical record, the first word of the first n-1 physical records contain zero; the first word of the nth physical record contains the integer n. This first word indicates how many physical records exist in a logical record. If there is only one physical record in the logical record, the first word contains the integer 1.

When end of tape is encountered during the writing of a logical record, the tape is repositioned to the beginning of the record and a flag is set which may be sensed by IF(EOF, i).

Examples:

```
      DIMENSION  A(260), B(4)
      WRITE(10)A,B
          writes 1 logical record of 2 physical records
      DO 5 I = 1, 10
  5   WRITE TAPE 6, AMAX(I), (M(I,J), J = 1, 5)
```

WRITE: BINARY(ODD PARITY)
k WORDS

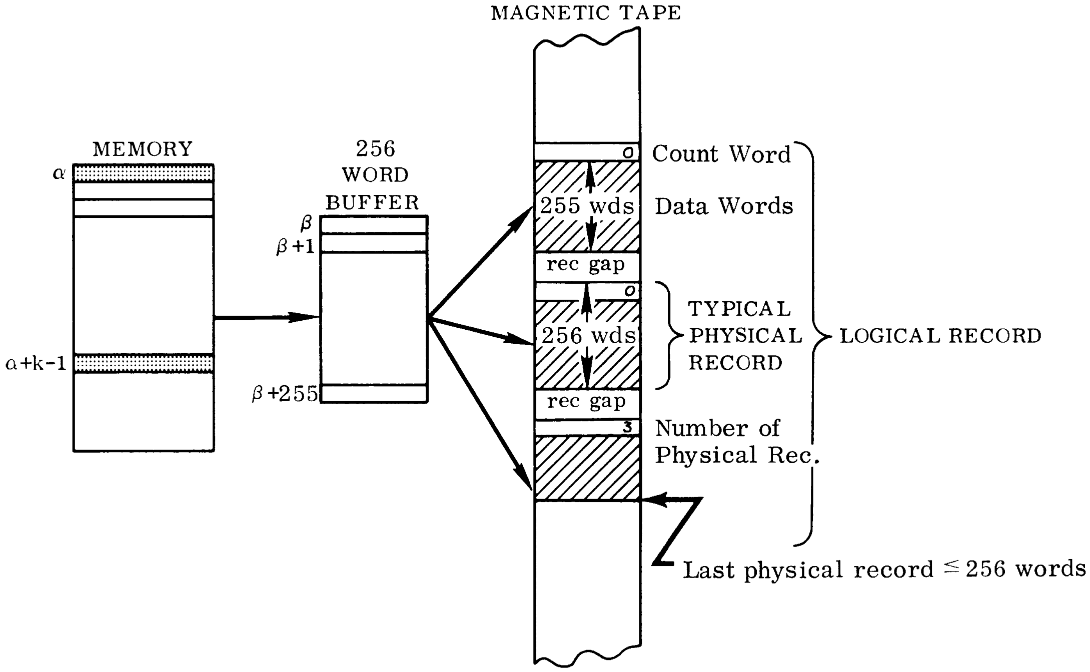


α represents a word in storage

β represents the first word of a physical record on tape

Figure 1a.

WRITE: BINARY(ODD PARITY)
k WORDS
MEMORY TAPE SCHEMATIC



EXAMPLE: Write 520 binary words on tape.

A. Set count to 1. First 255 words placed in buffer.
More words remain so first buffer word is 0.
Write 256 word physical record on tape.
Bump count 1.

B. Next 255 words to buffer. Same procedure as A.
Bump count 1.

C. 10 words remain. Transfer to Buffer;

Enter count (3) in first buffer word.
Write 11 word physical record on tape.
Exit.

Figure 1b.

9.1.2

READ STATEMENTS

READ n,L reads one or more card images, converting the information from left to right, in accordance with the FORMAT specification (n) and stores the converted data in the storage locations named by the list (L) identifiers. The images read may come from 80-column Hollerith cards, or from magnetic tapes, prepared off-line containing 80-character records in BCD mode. Note caution under BUFFER IN for intermixing READ n, L and BUFFER IN statements.

Example:

```
      READ  10, A, B, C
  10  FORMAT  (3F10.4)
```

READ (i,n)L and **READ INPUT TAPE i,n, L**
are equivalent forms which transfer one logical record of information from a specified logical unit (i), 1 through 50, to storage locations named by the list (L) identifiers according to FORMAT statement (n).

The number of words in the list and the FORMAT specifications must conform to the record structure on the logical unit (up to 120 characters in the BCD mode). A record read by READ (i,n)L should be the result of a BCD mode WRITE statement. A binary record read in BCD mode will produce a parity error. Note caution under BUFFER IN for intermixing READ (i,n)L and BUFFER IN statements.

Examples:

```
      READ INPUT TAPE 10, 11, X, Y, Z
  11  FORMAT (3F10.6)

      TYPE DOUBLE D2
      DIMENSION D2(4)
      READ (10, 12) D2
  12  FORMAT (4D25.16)

      READ INPUT TAPE 4,22
  22  FORMAT (33H ..........................)

      READ (2, 13) (Z (K), K = 1, 8)
  13  FORMAT (F10,4)
```

READ (i) L and **READ TAPE i,L**

are equivalent forms which transfer one logical record of information from a specified logical unit (i), 1 through 49, to storage locations named by the list (L) identifiers.

A record read by READ (i) should have been written in binary mode. The count word is not transmitted to the input area, L. The number of words in the list of READ (i) L must be equal to or less than the number of words in the corresponding WRITE statement.

If the list (L) is omitted, READ (i) spaces over one logical record.

Caution

If the record read by READ (i) L was written with a BUFFER OUT statement, the first word of each physical record is not transmitted.

Examples:

```
      DIMENSION  C(264)
      READ  (10)C

      DIMENSION BMAX (10), M2 (10, 5)
      DO7I=1,10
    7 READ TAPE 6, BMAX (I), (M2(I,J), J=1,5)

      READ  (5)          (skip one logical record on unit 5)

      READ  (6)  ( (A(I,J),I=1,100),J=1,50)

      READ TAPE 6, ( (A(I,J), I=1,100),J=1,50)
```

9.2 BUFFER STATEMENTS

There are three primary differences between the buffer I/O statements and the read/write I/O statements.

1. The mode of transmission (BCD or binary) is tacitly implied by the form of the read/write control statement. In a buffer control statement, parity must be specified by a parity indicator.
2. The read/write control statements are associated with a list, and, in BCD transmission, with a FORMAT statement. The buffer control statements are not associated with a list; data transmission is to or from one area in storage.

3. A buffer control statement initiates data transmission, and then returns control to the program, permitting the program to perform other tasks while data transmission is in progress. Before using any of the buffered data, the status of the buffer operation should be checked. See section 9.5. A read/write control statement completes the operation indicated before returning control to the program.

In the descriptions that follow, these definitions apply.

i	logical unit number: from 1 to 52 (integer constant or variable).
p	recording mode (integer constant or variable): 0 for BCD; 1 for row binary; 2 for column binary. The recording mode interpretations for magnetic tapes are: 0 selects even parity; 1 and 2 select odd parity. The interpretations for other I/O equipment are given in the CO-OP MONITOR/Programmer's Guide, where the Monitor mode is given by p + 1.†
A	variable identifier: first word of data block to be transmitted.
B	variable identifier: last word of data block to be transmitted.

A magnetic tape written in odd parity must be buffered in odd parity; a tape written in BCD mode must be buffered in even parity.

BUFFER IN (i,p) (A,B)

transmits information from unit i in mode p to storage locations A through B. The record structure is shown in figure 2. If a magnetic tape containing BCD records written by WRITE (i, n) is used by BUFFER IN, only one physical record (15 words or less), will be read. When a magnetic tape written by WRITE (i) is read by BUFFER IN, provision must be made for the count word which is buffered in with the transmitted data. Only one physical record is read for each BUFFER IN statement (figures 1a and 1b).

Caution

BCD read statement (READ n,L and READ (i,n)L) and BUFFER IN statements may both be used for input from the card reader. BCD reads will input one more record than specified by the statement. If a BUFFER IN statement follows a BCD read, to prevent the loss of a record, a dummy record should separate those specified in the BCD read from those to be buffered in.

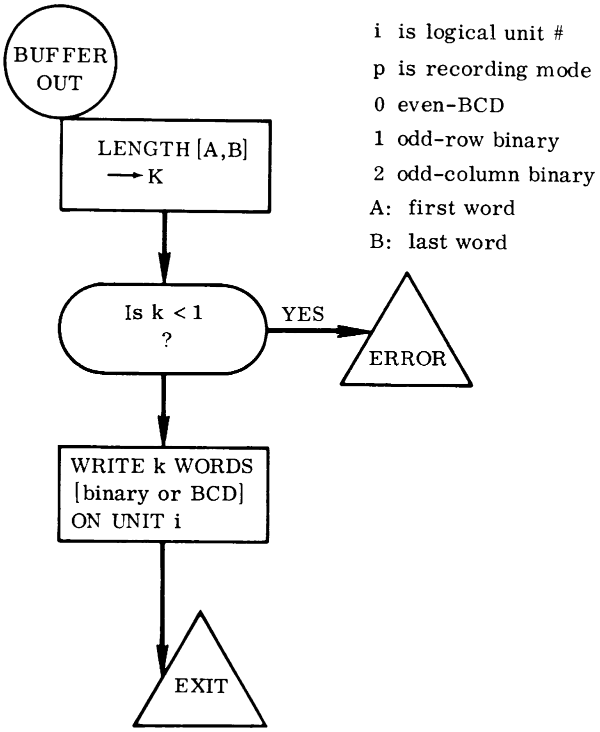
BUFFER OUT (i,p) (A,B)

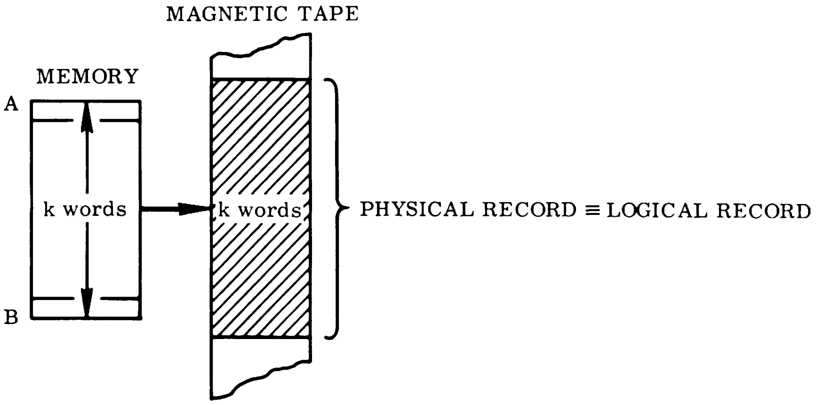
transmits information from storage locations A through B, and writes one physical record on logical unit i in mode p. The physical record contains all the words from A to B inclusive (figure 2).

---

†The function code (F.C.) is 1 with an interrupt (I) of 1 when buffering.

BUFFERED WRITE: BINARY OR BCD
BUFFER OUT (i,p) (A,B)



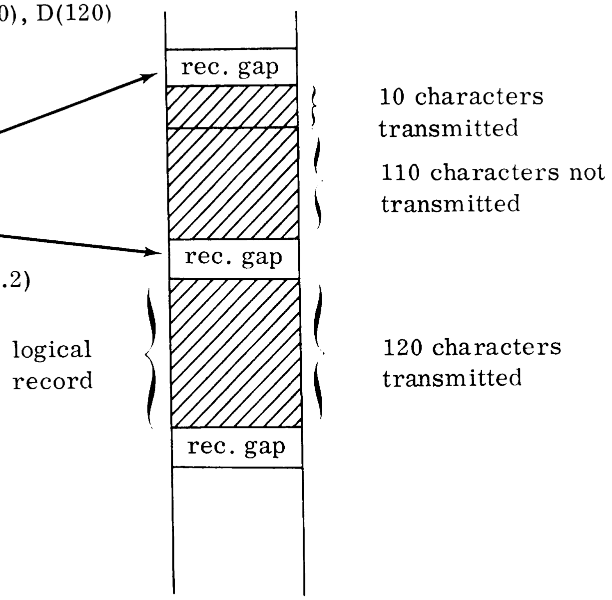


9.3 PARTIAL RECORD

The tape unit always moves to the next logical record after a READ(i, n) L, READ (i)L, or to the next physical record after a BUFFER IN statement, even if the entire record is not transmitted. Consequently, the remainder of the record will not be read with the next READ or BUFFER IN statement.

Example:

```
     DIMENSION C(10), D(120)
         .
         .
         .
     READ (3, 10) C
10   FORMAT (10A1)
     READ (3, 12) D
12   FORMAT (12F10.2)
```



9.4 TAPE HANDLING STATEMENTS

The logical unit number, i, may be an integer variable or constant.

REWIND i

rewinds the magnetic tape mounted on unit i to load point. If the tape is already rewound, the statement acts as a do-nothing statement. i may be 1 through 49.

BACKSPACE i

backspaces the magnetic tape mounted on unit i one logical record. (A logical record is a physical record; except for tapes written by a WRITE (i)L statement). If tape is at load point (rewound) this statement acts as a do-nothing statement. When backspacing on standard units 51 or 52, no more records may be backspaced than have been written. When backspacing on standard unit 50, no more records may be backspaced than have been read. i may be 1 through 52.

END FILE i

writes an end-of-file on the magnetic tape mounted on unit i, 1 through 49, 51 or 52.

9.5

STATUS CHECKING STATEMENTS

IF(EOF) and IF (IOCHECK) are the status checking statements to be used with the read/write I/O control statements.

IF(EOF,i)n_1,n_2

checks the previous read (write) operation to determine if an end-of-file (end-of-tape) has been encountered on unit i. If it has, control is transferred to statement n_1; if not, control is transferred to statement n_2.

IF(IOCHECK,i)n_1,n_2

checks the previous read (write) operation to determine if a parity error has occurred on unit i. If it has, control is transferred to statement n_1; if not, control is transferred to statement n_2.

IF(UNIT,i)n_1,n_2,n_3,n_4

is used with buffer control. To avoid loss of information, this statement should always appear before the first statement that uses any variables transferred in the buffer mode. The n_i are statement numbers. If any branch points are omitted, their error checks will not be made.

This statement checks the status of the last buffering operation on unit i and will transfer control to statement:

n_1	if buffer operation is not complete
n_2	if buffer operation is complete with no errors
n_3	if buffer operation is complete and an EOF or EOT occurred
n_4	if buffer operation is complete and a parity error occurred

When a parity error occurs, FORTRAN-63 will attempt to execute a BUFFER IN statement six times and a BUFFER OUT statement three times. Unit i will not be sensed ready until there is no parity error or until the number of repetitions has been exhausted. If an EOT and parity error occur simultaneously, only the EOT jump is made.

LENGTH (i) FUNCTION

is used with an integer variable, for example I=LENGTHF (i), to find the number of 48-bit words read during the last input operation on unit i. It may be used only with the BUFFER IN statement and must be preceded by an IF(UNIT, i) statement to insure that the input is completed; there may not be an intervening buffer statement regardless of the logical unit number.

Example:

```
                PROGRAM                               REMARKS

         J=1                                   Set flag =1

         BUFFER IN (10, 0) (A, Z)              Initiate buffered read in even (BCD)
                                               parity.

4        IF (UNIT, 10)5, 6, 7, 8               Check status of buffered transfer.
5        GO TO (50, 4), J                      Not finished.  Do calculations at 50.

50     { Some computation not involving }
       { information in locations A - Z }

         J=J+1                                 Calculations complete; increase
         GO TO 4                               flag by 1.  Go to 4.

7        PRINT 70
70       FORMAT (12H EOF UNIT 10)              End of file error
         GO TO 200

8        PRINT 80

80       FORMAT (35H PARITY OR BUF LENGTH ERROR UNIT 10)


200      REWIND 10                             Rewind tape and stop

         STOP                                  Stop

6        CONTINUE                              Buffer transmission complete
             .                                 Continue program
             .
             .
```

9.6

ENCODE/DECODE STATEMENTS

The ENCODE/DECODE statements are comparable to the WRITE/READ statements with the essential difference being that no peripheral equipment is used in the data transfer. Information is transferred under FORMAT specifications from one area of storage to another.

In the following descriptions:

n	is a statement number, a variable identifier or a formal parameter representing the associated FORMAT statement.
L	is the input/output list.

V is a variable identifier or an array identifier which supplies the starting location of the record. The identifier may have standard or non-standard subscripts.

c is an unsigned integer constant or an integer variable (simple or subscripted) specifying the length of the record. c may be an arbitrary number of BCD characters. The record starts with the leftmost character of the location specified by V and continues c BCD characters, 8 BCD characters per computer word. Each record begins with a new computer word.

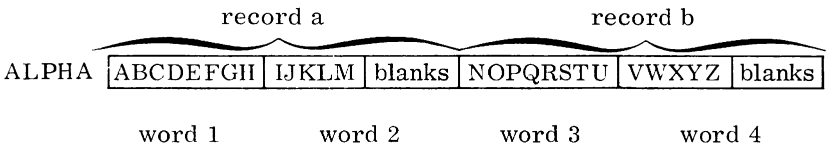
For ENCODE, if c is not a multiple of 8, the record ends in the middle of a computer word and the remainder of the word is blank-filled. For DECODE, if the record ends in the middle of a computer word, the remaining characters in that word are ignored.

Examples:

A(1) = ABCDEFGH
A(2) = IJKLM
B(1) = NOPQRSTU
B(2) = VWXYZ

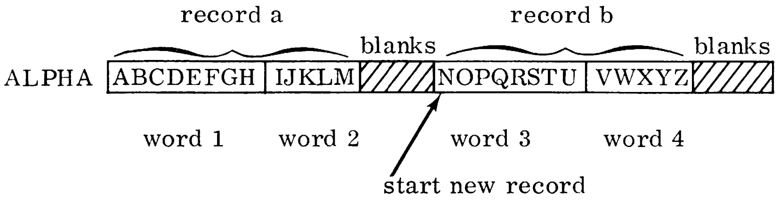
1) c=multiple of 8

```
    ENCODE (16, 10, ALPHA) A,B
10  FORMAT (2(A8, A5) )
```



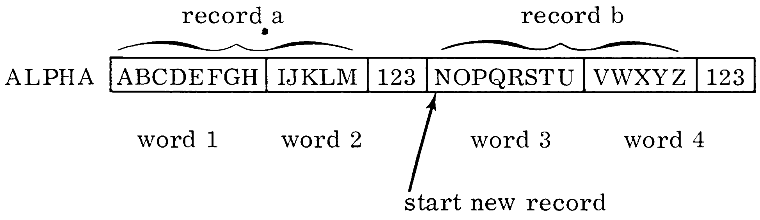
2) c≠multiple of 8

```
    ENCODE (13, 10, ALPHA)A, B
10  FORMAT (2(A8, A5) )
```



3) $c \neq$ multiple of 8

```
    DECODE (13, 10, ALPHA)A, B
10  FORMAT (2(A8, A5) )
```



ENCODE (c,n,V)L

transmits machine-language elements in a manner similar to PRINT n, L and PUNCH n, L. The information of the list variables, L, is transmitted according to the FORMAT (n) and stored in locations starting at V, c BCD characters per record. If the I/O list (L) and specification list (n) translate more than c characters, an execution time diagnostic, ERROR IN BCD OUT WIDTH, occurs. If the number of characters converted is less than c, the remainder of the record is filled with blanks.

DECODE (c,n,V) L

transmits and edits BCD characters in a manner similar to READ n, L. The information in c consecutive BCD characters (starting at address V) is transmitted according to the FORMAT (n) and stored in the list variables (L). If the number of characters specified by the I/O list and the specification list (n) is greater than c (record length), an execution time diagnostic occurs. If DECODE attempts to process an illegal BCD code or a character illegal under a given conversion specification, an execution time diagnostic, ERROR IN BCD IN DATA, occurs.

In ENCODE and DECODE, the record is an integral number of computer words, i.e. $(C + 7)/8$ words long.

Examples:

1) The following is one method of packing the partial contents of two words into one word. Information is stored in core as follows:

```
LOC(1)  SSSSxxxx
   .
   .
   .
LOC(6)  xxxxαααα
        8 bcd ch/wd
```

To form SSSSαααα in storage location NAME:

```
        DECODE(8,1,LOC(6) )TEMP
    1   FORMAT(4X,A4)
        ENCODE(8,2,NAME) LOC(1),TEMP
    2   FORMAT(2A4)
```

The DECODE statement places the last 4 BCD characters of LOC(6) into the first 4 characters of TEMP. The ENCODE statement packs the first 4 characters of LOC(1) and TEMP into NAME.

A more straightforward way of accomplishing this is with the R specification; the program may be shortened to:

```
        ENCODE (8,1,NAME) LOC(1),LOC(6)
    1   FORMAT (A4,R4)
```

2) DECODE may be used to calculate a field definition in a FORMAT specification at object time. Assume that in the statement FORMAT (2A8,Im) the programmer wishes to specify m at some point in the program, subject to the restriction $2 \leq m \leq 9$. The following program permits m to vary.

```
        IF(M  .LT. 10 .AND. M .GT. 1)1,2
    1   ENCODE (8,100,SPECMAT) M
  100   FORMAT (6H(2A8,I,I1,1H) )
        .
        .
        .
        PRINT SPECMAT,A,B,J
```

M is tested to insure it is within limits. If not, control goes to statement 2 which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters: (2A8,I). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A8,Im).

The print statement will print A and B under specification A8, and the quantity J under specification I2, or I3 or . . . or I9 according to the value of m.

3) ENCODE can be used to re-arrange and change the information in a record. The following example also shows that it is possible to encode an area into itself and that encoding will destroy information previously contained in an area.

```
       PROGRAM  ENCO2
       I=7RBCDEFGH
       IA=1H1
       ENCODE (7, 10, I)I, IA, I
   10  FORMAT (A2, A1, R4)
       PRINT 11, I
   11  FORMAT (O20)
       END
```

```
       PRINT OUT
                62016566677020
```

The BCD equivalent is

```
                B1EFGHblank
```

4) In this example, accounting information is to be read from a magnetic tape prepared off-line from 80-column Hollerith card input. Each record on this tape will be 10 words (80 characters) long. The program is to initiate a read, decode the information of this read and initiate a second read while decoding the information obtained from the first read. Two 10-word buffers are used (AIN and CIN). The FORMAT specification in DECODE is

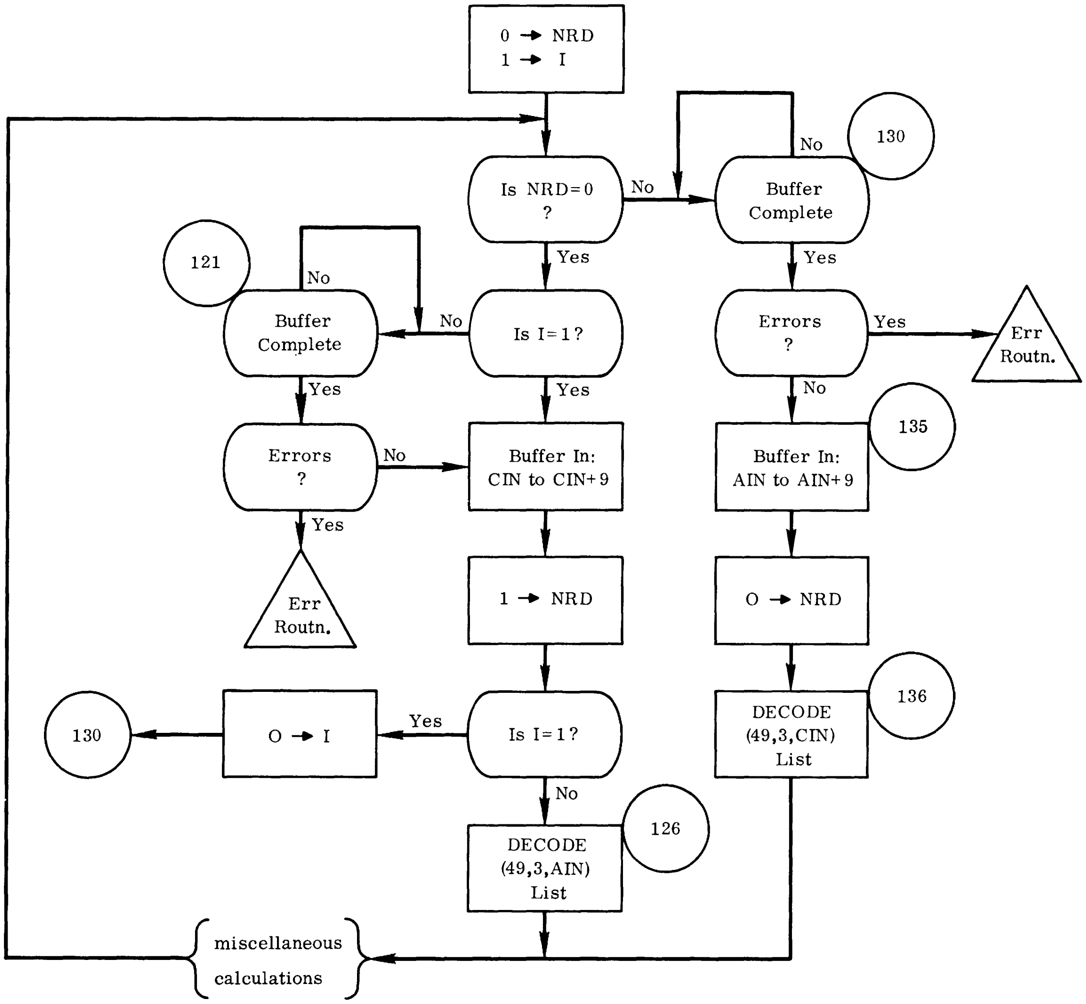
```
(6A1,A1,8A1,A3,I2,A6,4I2,2A1,A8,A3,2A1)
```

this specification breaks the first 49 characters of each BCD record read from magnetic tape. Let the list be the string of identifiers:

```
LIST:  DT,CC,CN,PR,X,XM,N1,M1,N2,M2,CR,ADJ,PER,RUN,ATT
```

DT is an array of length 6; CN is an array of length 8; the remaining identifiers name simple variables.

Flow chart of the basic procedure:



COMPILATION AND EXECUTION 10

1604 FORTRAN-63 source programs are compiled and executed under the CO-OP Monitor System.[†] The monitor controls job processing, equipment assignments, and input/output operations; it also provides debugging aids, error dumps, and diagnostics (Appendix H).

The monitor system loads the FORTRAN compiler into memory and transfers control to it. The compiler translates FORTRAN statements into CODAP-1 assembly language instructions, supplies diagnostics for source language errors, and directs the assembler to produce relocatable binary object programs which consist of binary card images on magnetic tape. The object program may be executed immediately or it may be saved on magnetic tape or punched onto cards to be executed at a later time.

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

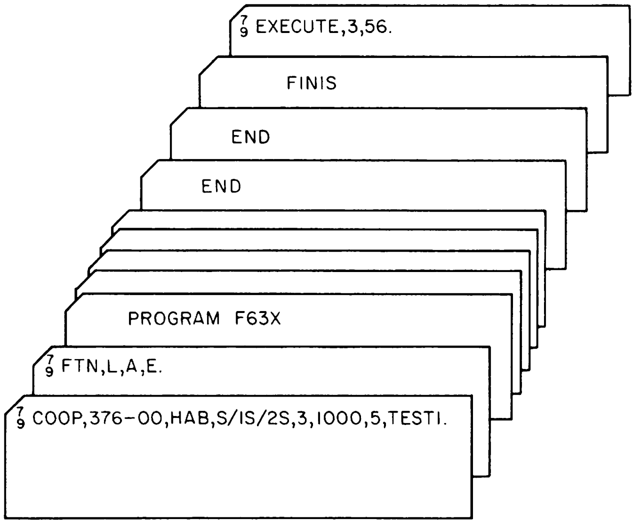
a) If a blank card appears between a statement and its continuation, the continuation and subsequent continuations are lost. Compilation continues.

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

[†] For the CO-OP control cards, see CO-OP MONITOR/PROGRAMMER'S GUIDE, publication number 60050800a.

10.1
CONTROL CARDS

A programmer sets up a deck for compilation or execution with a Master Control card, a FORTRAN control card, and various combinations of END, FINIS, EXECUTE and BINARY control cards properly placed. The figure below illustrates the control card arrangement for compilation and execution (load-and-go) of a FORTRAN-63 program. The Master Control (MCS) card is first followed by the FORTRAN control card, which in this case specifies FORTRAN-63. Next is the source program deck with two FORTRAN END cards. Each END card will be compiled as a transfer card; two successive transfer cards are required to terminate the loading procedure. The FINIS and EXECUTE cards follow. A data deck may follow the EXECUTE card.



LOAD-AND-GO

10.1.1
MCS CARD

$^7_9$COOP, A, I, IO, TL, LL, R, C.

Provides accounting information for the operations center, establishes time and line output limits for the job, provides tape assignment information and specifies recovery procedures in case of abnormal termination.

Field 1	7-9 punch in column 1 followed immediately by COOP specifies the Monitor system.
Field 2 (A)	Accounting information
Field 3 (I)	Programmer's initials
Field 4 (IO)	I/O assignment field (Section 3.2, Appendix F). 2 S should be specified if there is a possibility that any subprogram may exceed the compiler's available core capacity. This tape is used as scratch by the compiler to hold excess assembly code prior to assembly.
Field 5 (TL)	Time limit estimate in minutes. If not sufficient, job is terminated before completion. For compilation, assume a rate of 125 source language cards per minute. If the time limit is exceeded the recovery procedure is followed.
Field 6 (LL)	Line limit estimate. This number should be greater than the maximum number of output lines anticipated, including compilation listings. If it is less, the job is terminated before completion.
Field 7 (R)	(optional) Recovery key indicates recovery (dump) procedure.
Field 8 (C)	(optional) Comments or identification

A comma follows each field except the last which is followed by a period. The card is free field after column 2. Up to 8 cards may be used if necessary; each card must have a 7-9 punch in column 1, and a Hollerith character in column 2.

Example:

${}^{7}_{9}$COOP, 347-00, JSM, S/1S/2S, 10, 1500, 5, COMTEST.

If an omitted field is followed by another field, the comma rule must be observed. For example, if scratch unit assignments are not made the MCS card may read: ${}^{7}_{9}$COOP, 347-00, JSM, , 10, 1500, 5, COMTEST.

10.1.2
FORTRAN CARD

${}^{7}_{9}$FTN, options.

Loads the FORTRAN system.

Field 1	7-9 punch in column 1 followed immediately by FTN specifies FORTRAN-63.

The card is free field after column 2. The options may appear in any order separated by commas. Unrecognized options and extraneous characters are ignored. The option field is terminated by a period at the end of the control card. If no options are present, only error messages and the basic assembler headings are printed. Any option can be abbreviated to its first character only, $^{7}_{9}$FTN, L, E, B. Any option may be followed by = n, $^{7}_{9}$FTN, LIST=1, E=10.

Options:		$n \neq 0$
LIST	List source language program on 51	List source language program
PUNCH	Punch relocatable binary deck on logical unit 52	Punch binary on unit n.
EXECUTE	Write load-and-go tape 56	Write load-and-go tape n
ASSEMBLY	List assembled programs in CODAP1 language on 51	List assembled programs in CODAP1
INPUT	Input source from 50. Same even if option is not present	Input source from n
TAPE	No assembler scratch tape; same if option is omitted	Assembler scratch tape n
BCD	Punch generated CODAP1 cards on 52	Punch generated cards on n
SYMBOLS	Allot 2048 words to Assembler Symbol Table; if option is omitted, allot 1024 words	Allot (max. n, min. 1024) words to Assembler Symbol Table
REFERENCES*	Suppress Assembler Symbol Table; if option is omitted, print table	Suppress Table
NULLS	Suppress Null listing; if option is omitted, print Null listing.	Suppress Null listing

If n is 0, the option is interpreted as if it were not present

10.1.3
FINIS CARD

FINIS

Indicates compilation is to end; it is used only in conjunction with compilation. The word begins in column 10.

---

*Applies only if ASSEMBLY option is present.

10.1.4
EXECUTE CARD

${}^{7}_{9}$EXECUTE, TL, LGU, SL.

When EXECUTE precedes a relocatable binary deck (RBD), the program from the standard input unit (3.1) is loaded into core. When EXECUTE accompanies a load-and-go tape, the program from the specified unit is loaded into core and executed. (See repeated job execution with N data decks and batch execution and partial compilation and execution - 2.2, 2.3, 2.5.)

Field 1	7-9 punch in column 1 followed immediately by the word EXECUTE.
Field 2 (TL)	Time limit of execution in minutes. If not sufficient, job is terminated before completion. If greater than the time limit on MCS card, it is ignored.
Field 3 (LGU)	Load-and-go unit. If omitted or blank with load-and-go, unit 56 is assigned. It must agree with the corresponding assignment on the MCS and FORTRAN cards.
Field 4 (SL)	Suppress map listing key; 1 if a listing is not desired, otherwise omitted.

A comma follows each field except the last which is followed by a period. The card is free field after column 2, embedded blanks may be used and field lengths are variable.

Example:

${}^{7}_{9}$EXECUTE, 3, 10, 1.

Execute program from load-and-go unit 10 with a time limit of 3 minutes; suppress map listing.

10.1.5
BINARY CARD

${}^{7}_{9}$ BINARY, N.

Transfers binary card images from the standard input unit to unit N until a control card is encountered.

Field 1	7-9 punch in column 1 followed immediately by the word BINARY.
Field 2 (N)	Logical unit designator. N is an integer, 1 to 49 or 56. If N is blank or omitted, it is assumed to be unit 56. It must agree with the corresponding assignment on the MCS, FORTRAN and EXECUTE cards.

The card is free field after column 2.

Example:

$^{7}_{9}$BINARY, 56. or $^{7}_{9}$BINARY.

The binary card images which follow will be transferred from the standard input unit (50) to the standard scratch unit (56).

$^{7}_{9}$BINARY, 5.

The binary card images which follow will be transferred from the standard input unit to logical unit 5.

10.1.6

FORTRAN-63 SOURCE DECK

This deck contains the program and all its subroutines except those from the Library. The program may contain assembly (CODAP1) language subprograms and FORTRAN-63 subprograms in any order after the FORTRAN card. (The presence of CODAP1 subprograms in the source deck does not require a CODAP card.)

10.2
DECK STRUCTURE

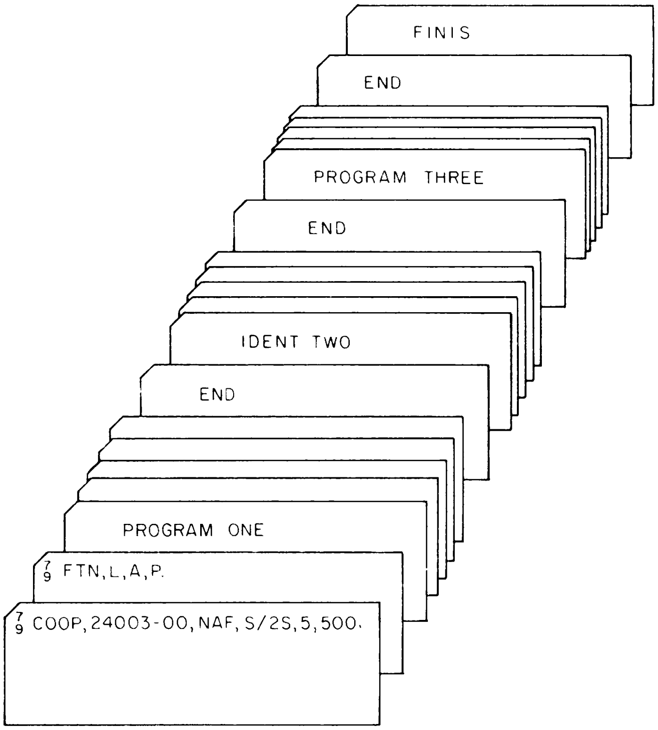
10.2.1
COMPILATION ONLY

Compile one or more FORTRAN-63 programs or subprograms.

Deck Structure:

1.	MCS card	scratch unit 2S must be assigned as an overflow scratch unit
2.	FORTRAN card	omit load-and-go assignment
3.	Source decks	(Source Programs - FORTRAN-63 and/or CODAP1)
4.	FINIS card	

In the figures in this section, the END cards in the source decks represent the terminal END cards existing with the source programs.



BATCH COMPILATION

a) ${}^{7}_{9}$COOP, 24003-00, NAF, S/2S, 5, 500.

Scratch units assigned; alternate form for assignment is S/57. Time limit is 5 minutes. Line limit is 500 lines.

b) ${}^{7}_{9}$FTN, L, A, P.

List source and assembly language versions and punch the binary deck.

c) FORTRAN-63 and CODAP1 source language subprograms. Required END cards must be in place after each subprogram.

d) FINIS card to signal end of compilation begins in column 10.

For batch compilation, stack the source decks sequentially each with its END card. One FINIS card appears after the last deck to be compiled.

10.2.2

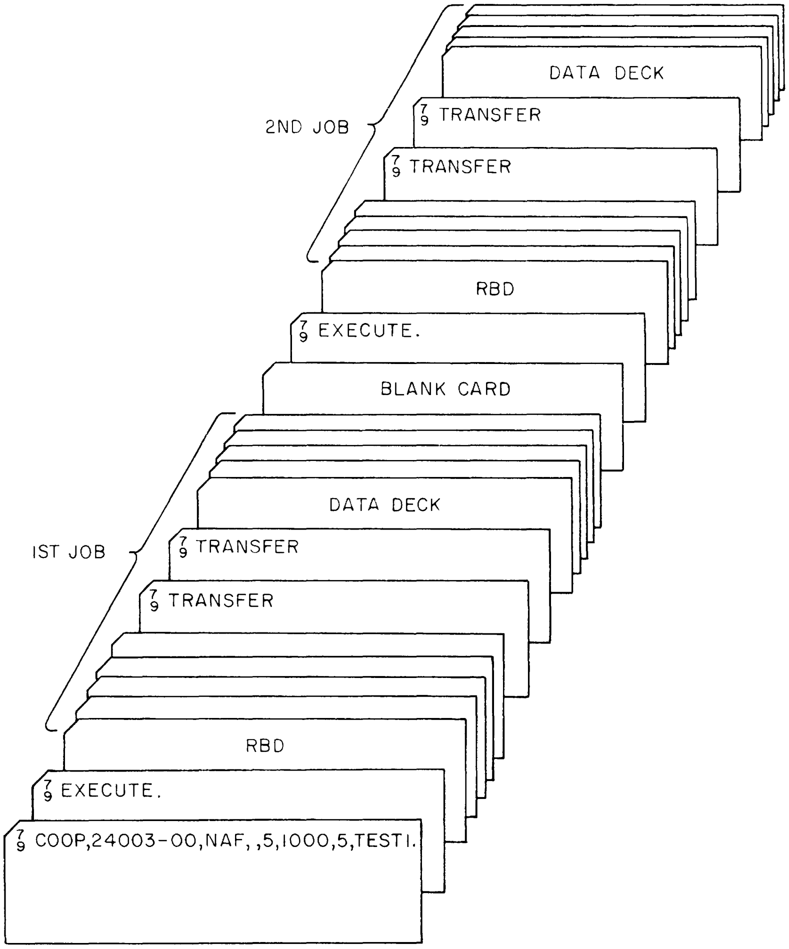
EXECUTION ONLY

Single Job and Multiple Job.

To execute a compiled program with or without data.

Deck Structure:

1. MCS card
2. EXECUTE card
3. Relocatable Binary Deck
4. Transfer cards (card containing only 7-9 punch, col 1)
5. Data deck if applicable



BATCH EXECUTION

a) $^7_9$COOP, 24003-00, NAF, , 5, 1000, 5, TEST1.

Scratch units and other I/O units are not required. If used they appear in field four. Time limit is 5 minutes;line limit is 1000. For abnormal job termination, perform recovery procedure 5 (3.1.1).

b) $^7_9$EXECUTE.

Execute the program with the time limit specified.

c) Relocatable Binary Deck (Object Program)

If the deck has two transfer cards, go to step d. The RBD will have two transfer cards only if the source deck was terminated with an extra FORTRAN END card. If a second END was not included in the source deck, there will be only one transfer card generated in the RBD, and a second transfer card must be provided by the programmer.

d) Data cards complete the deck set-up.

Batch executions are set up as above for the first job; subsequent jobs are preceded by a blank card followed by an EXECUTE card.

10.2.3

EXECUTION ONLY

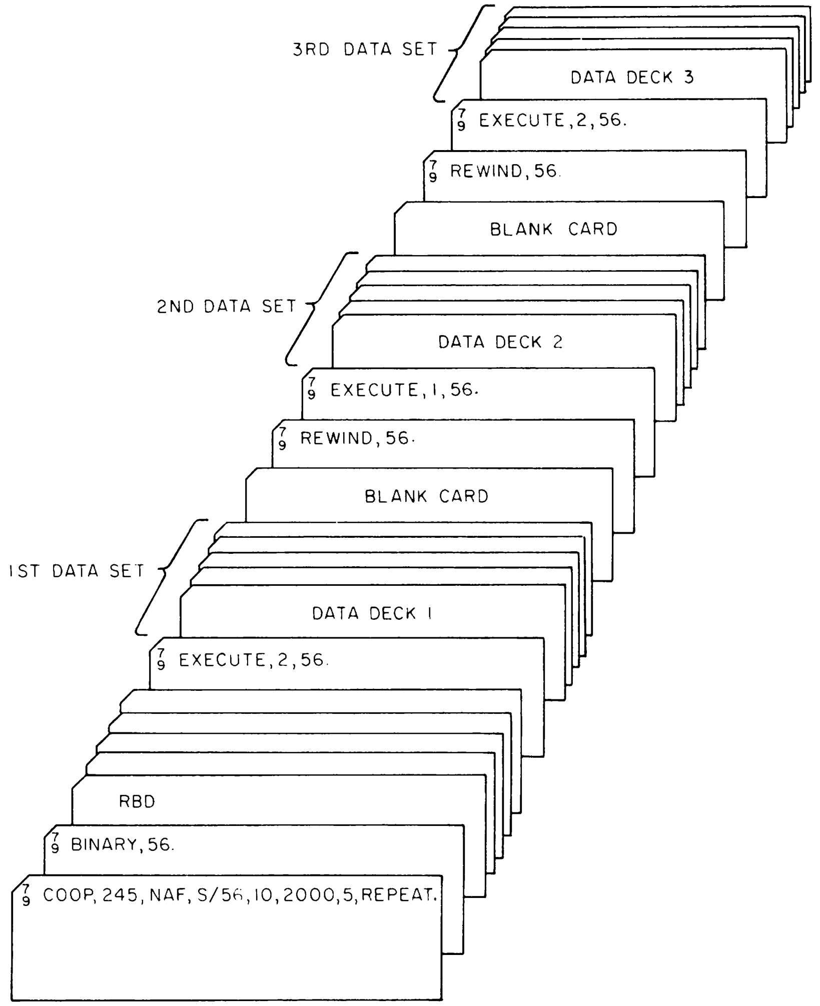
Repeated execution of one RBD with N data sets.

To execute a program with more than one set of data.

Deck structure:

1. MCS card
2. BINARY card
3. Relocatable Binary Deck
4. EXECUTE card
5. Data deck
6. Blank card
7. REWIND card
8. EXECUTE card
9. Data deck

   (repeat steps 6, 7, 8 as required.)



ONE JOB, THREE DATA DECKS

a) $^{7}_{9}$COOP, 245, NAF, S/56, 10, 2000, 5, REPEAT.
Time limit is 10 minutes. Line limit is 2000. Recovery procedure 5.

b) $^{7}_{9}$BINARY, 56. or $^{7}_{9}$ BINARY.
Scratch unit 56 designated for RBD.

c) Relocatable binary deck. RBD must have two terminal transfer cards (7-9 punch, col. 1)

d) $^{7}_{9}$EXECUTE, 2, 56. or $^{7}_{9}$ EXECUTE, 2.
Ready for execution with first set of data. Time limit is 2 minutes. Scratch 56 need not be specified; it is assumed if omitted.

e) Data deck for first execute.

f) Blank card

g) ${}^{7}_{9}$REWIND, 56.

h) ${}^{7}_{9}$EXECUTE, 1, 56.

Ready to execute next set of data. Time limit 1 minute. Load-and-go unit must be specified.

i) Data deck for second execute.

j) Blank card

.
.
.

Total of individual execution times must not exceed total time specified on the MCS card.

10.2.4

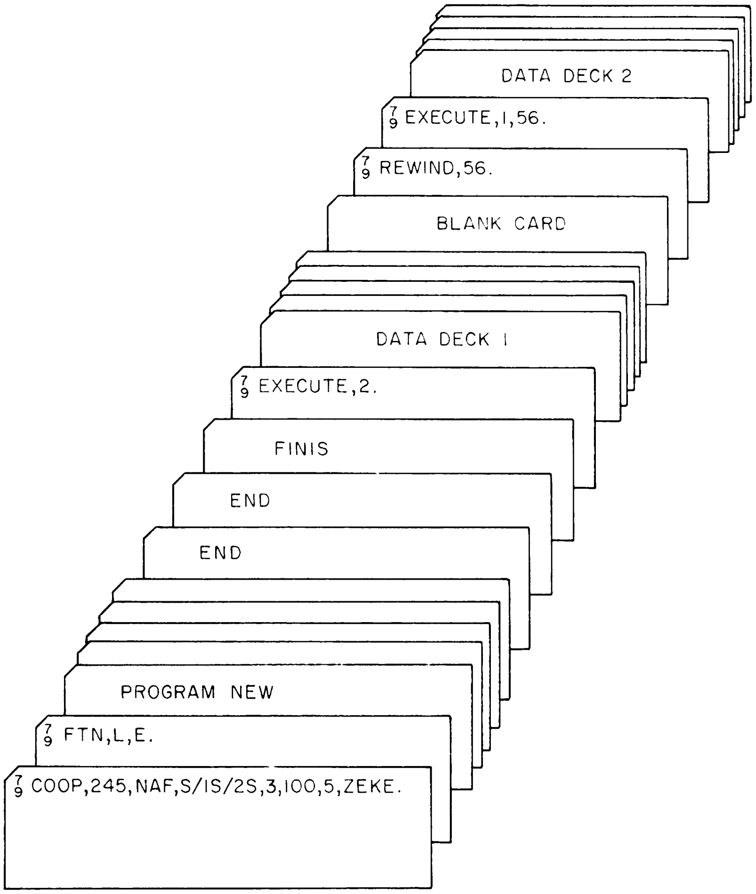
COMPILATION AND EXECUTION

Load-and-go

To compile a FORTRAN program and execute it immediately with or without data.

Deck structure:

1. MCS card
2. FORTRAN card
3. Source deck, 2 END cards
4. FINIS card
5. EXECUTE card
6. Data deck



COMPILE AND EXECUTE (LOAD-AND-GO) WITH TWO DATA DECKS

a) $^7_9$COOP, 245, NAF, S/1S/2S, 3, 100, 5, ZEKE.

Scratch units required for compilation. Total time for compilation and execution is 3 minutes. Line limit is 100. Recovery procedure 5. Load-and-go unit is 1S (unit 56).

b) $^7_9$FTN, L, E.

Provide listings. No RBD. Load-and-go unit 56.

c) Source deck

Two terminal FORTRAN END cards will generate two terminal transfer cards (7-9 punch, col. 1). If only one terminal FORTRAN END card is used, a transfer card must be inserted immediately after the EXECUTE card. The deck may also be CODAP1 with two terminal END cards.

d) FINIS card

Signals end of compilation

e) ${}^{7}_{9}$EXECUTE, 2, 56. or ${}^{7}_{9}$EXECUTE. 2.

Execute the program with a time limit of 2 minutes. The time limit here must be less than the time limit on the MCS card. If compilation took 1.5 minutes, the job will be terminated after the remaining 1.5 minutes elapses.

f) Data deck

For repeated executions with data deck, repeat steps f through i section 2.3, Execution Only.

10.2.5
PARTIAL COMPILATION AND EXECUTION

To recompile a subroutine, or add a subroutine to an existing RBD and execute immediately, with or without data.

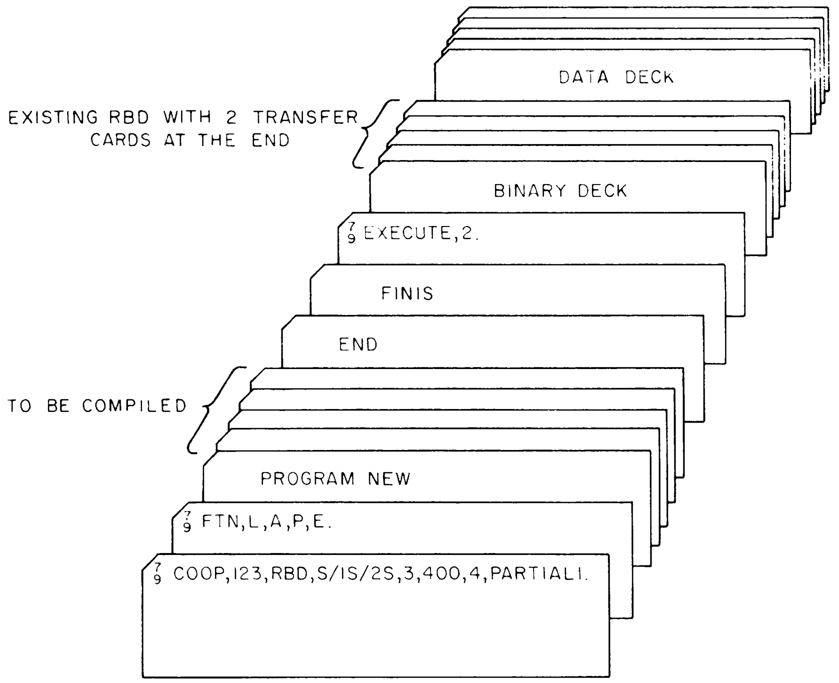
Procedure I loads the subprogram to be compiled before the existing RBD; execution then takes place. Procedure II loads the existing RBD, then the newly compiled subprogram.

PROCEDURE I

Procedure I must be followed when a special subroutine is to be used instead of an existing FORTRAN-63 library function with the same name. For example, in a program, LOGF might be the programmer's own function subroutine. To make certain his routine, and not the library LOGF is used, Procedure I is followed.

Deck Structure:

1. MCS card
2. FORTRAN card (FTN card)
3. Source deck (to be compiled)
4. FINIS
5. EXECUTE card
6. Relocatable Binary Deck (existing RBD)
7. Data deck



PARTIAL COMPILATION AND EXECUTION: PROCEDURE 1

a) $^{7}_{9}$COOP, 123, RBD, S/1S/2S, 3, 400, 4, PARTIAL1.

Scratch units assigned for compilation. Time limit is 3 minutes. Line limit is 400. Recovery procedure 4. Load-and-go tape is 1S (unit 56).

b) $^{7}_{9}$FTN, L, A, P, E.

Provide listing and RBD from compilation. Scratch unit 56 is load-and-go tape.

c) Source Deck (FORTRAN-63 or CODAP1)

Contains 1 terminal END card. Compilation will use unit 56.

d) FINIS

Signals end of compilation.

e) EXECUTE, 2.

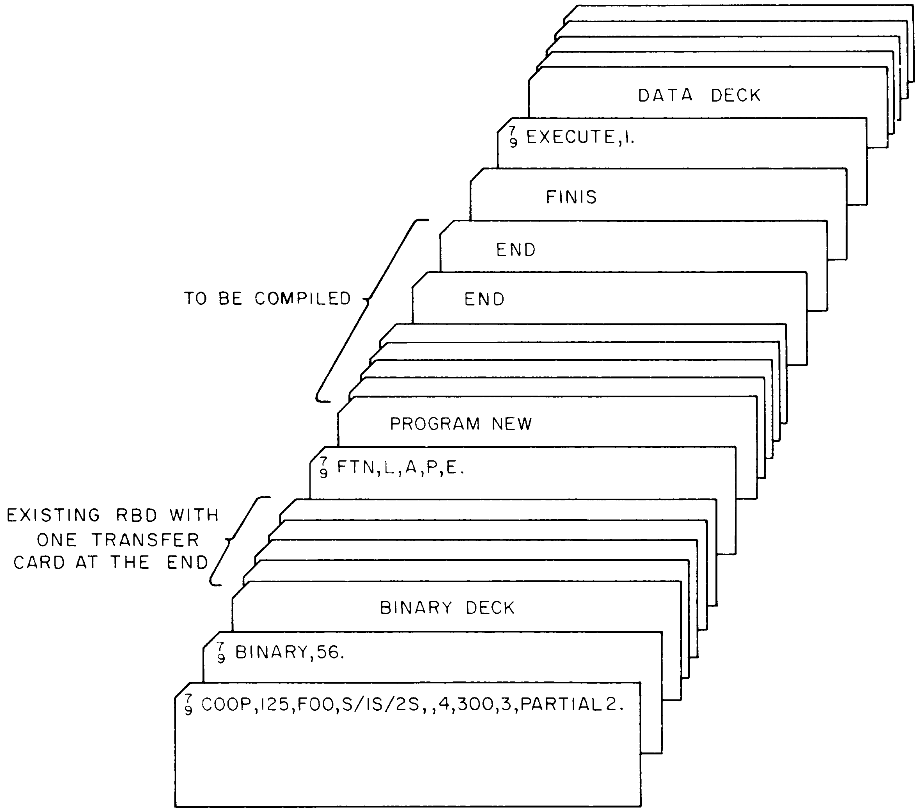
Time limit is 2 minutes. Unit 56 is assumed load-and-go unit. The newly compiled program and the RBD will be loaded into core in that order and executed.

f) Relocatable Binary Deck with 2 terminal transfer cards. (7-9 punch, col. 1)

g) Data deck

PROCEDURE II Deck Structure:

1. MCS card
2. BINARY card
3. Relocatable Binary Deck (existing RBD)
4. FORTRAN card (FTN card)
5. Source deck (to be compiled)
6. FINIS card
7. EXECUTE card
8. Data deck



PARTIAL COMPILATION AND EXECUTION: PROCEDURE 2

a) ${}^{7}_{9}$COOP, 125, FOO, S/1S/2S, 4, 300, 3, PARTIAL2.

Scratch units assigned for compilation. Time limit is 4 minutes; line limit is 300. Recovery procedure 3. Load-and-go tape is 1S (unit 56).

b) ${}^{7}_{9}$BINARY, 56. or ${}^{7}_{9}$BINARY.

RBD to be transferred to unit 56.

c) Relocatable Binary Deck (existing RBD)

d) ${}^{7}_{9}$FTN, L, A, P, E.

Listing and RBD compilation are required. Scratch unit 56 is load-and-go tape.

e) Source deck (FORTRAN-63 or CODAP1)

Assume two END cards appear. If there is only one, a transfer card (7-9 punch, col. 1) must be inserted immediately following the EXECUTE card.

f) FINIS card

Signals end of compilation.

g) ${}^{7}_{9}$EXECUTE, 1.

Time limit is 1 minute. Unit 56 is assumed load-and-go unit.

h) Data deck

10.3
INPUT/OUTPUT EQUIPMENT USAGE

When a FORTRAN-63 job is loaded for execution, the monitor assigns physical units corresponding to the logical units used by the program. Of all the units connected to the computer, a subset, called standard units, are assigned by the monitor for its own use. The standard units are assigned automatically and the user need be concerned only with the standard scratch units (56 or 1S. 57 or 2S). These are assigned by the user on the MCS control card when compilations are made. Logical unit 57 is used as an intermediate scratch unit by the source language processor. Logical unit 56 is assumed to be the load-and-go unit by the control system unless otherwise specified on the EXECUTE card.

10.3.1
STANDARD I/O UNITS

Standard Input Unit

This unit handles the system input requirements. Control cards, source programs, object decks for loading and input required by a FORTRAN READ n, L statement are read from this unit.

Standard Output Unit

This unit handles the system listable output requirements. Control information, listings, dumps, and output for a FORTRAN PRINT statement are written on this unit.

Standard Punch Unit

This unit handles the system punched card output requirements. Source language processor output (RBD), and output for a FORTRAN PUNCH statement are written on this unit.

The standard units and recovery key options are listed below.

Name	Unit #	Remarks
Standard Input Unit	50	
Standard Output Unit	51	
Standard Punch Unit	52	
Comment from Operator	53	typewriter
Comment to Operator	54	typewriter
Accounting Unit	55	paper tape
Standard Scratch Unit 1	56	also 1S
Standard Scratch Unit 2	57	also 2S

Recovery Key Option	Recovery Action Taken
0 or blank	Octal dump of console conditions on standard output unit
1	Same as 0 plus octal dump of numbered common region
2	Same as 0 plus octal dump of labeled common and the program
3	Same as 0 plus octal dump of labeled and numbered common and the program
5	Same as 0 plus octal dump of all of memory
4	Same as 5 except monitor regions of memory are not dumped

10.3.2 INPUT/OUTPUT FIELD OF THE MCS CARD*

Field 4 of the MCS card is the I/O unit assignment field. The following typical entry for this field assigns logical units 3 and 4 as input units and logical unit 5 as an output unit.

I/3/4/O/5,

The general form of field 4 is:

$$I/i_1/i_2/\ldots/i_p/O/o_1/o_2/\ldots/o_j/S/s_1/s_2/\ldots/s_m/E/\ell_1=p_1/\ell_2=p_2/\ldots$$

i_i, o_i, s_i, ℓ_i are logical unit numbers - The ranges are: 1 to 49

s_i may also be 56 (1S) or 57 (2S)

p_i is a logical unit number previously defined in the I/O list, or a standard I/O unit number.

I Logical units in the I list are assigned as input units; an input unit may also be assigned as an output unit.

O Logical units in the O list are assigned as output units; an output unit may also be assigned as an input unit.

S Logical units in the S list are assigned as scratch units and may function as both input and output units.

E Logical units in the E (Equivalence) list on the left hand side (the ℓ_i) of the = sign will be assigned to the same physical unit as the logical units on the right hand side (the p_i) of the = sign.

---

*Described in COOP MONITOR Programmer's Guide.

The order of assignments of I/O units is: input, output, scratch, equivalence. If only output and scratch units are assigned, they appear as

$$O/o_1/S/s_1/s_2$$

Assignments should not be made in any other order. If a unit is defined for one operation and an attempt is made to use it for another operation, the program will terminate abnormally.

Examples of MCS Field 4 Assignments

Output only	O/10/12	
Input only	I/4/6/17	
Input and Output	I/5/10/O/3/4	
Scratch only	S/1/10	
	S/1S/2S	
Input, Output and Equivalence		
	I/3/O/5/E/3 = 50/5 = 51	
	input unit 3;	equated to standard input
	output unit 5;	equated to standard output

OVERLAYS AND SEGMENTS 11

Programs that exceed available memory may be divided into independent parts. Such programs consist of a main subprogram (which remains in core storage during execution), overlays of the main subprogram and segments of overlays. The main subprogram will call each overlay into memory and transfer control to it. An overlay may call an associated segment into memory or return control to the main subprogram. The main subprogram, one overlay and one segment may be in core storage at any time. An overlay may not call another overlay nor may a segment call another segment.[†]

An overlay may reference entry points and common blocks within the main subprogram. A segment may reference entry points and common blocks within the main subprogram or its controlling overlay. The main subprogram may reference neither entry points nor common blocks within overlays or segments, nor can an overlay reference these items in a segment.

A FORTRAN source program, consisting of a main subprogram and one or more overlays and segments, may be compiled and executed on a load-and-go basis or it may be compiled for later execution. The procedure for load-and-go involves compiling and/or loading the job on the load-and-go tape in relocatable binary form and then writing the job on the overlay(s) in absolute.

An overlay tape is composed of two or more absolute binary records, the last of which is terminated by two end-of-files. Each record, constituting a main, overlay, or segment subprogram, may contain many subprograms.

[†] For more detailed information concerning overlays and segments, refer to publications INSTANT CO-OP MONITOR, F60056100, and CO-OP MONITOR/PROGRAMMER'S GUIDE, number 60050800, Rev. A.

Rules:

1 Overlays and segments must be written as closed subprograms entered by return jump instructions.

2 Parameters may be transmitted from a main program to an overlay and from an overlay to any of its segments.

3 Overlays are numbered sequentially, starting at 1, on each overlay tape. Segments are numbered sequentially, starting at 1, for each overlay.

4 A maximum of four overlay tapes may be generated for one program.

5 A TRA card will be generated when compiling a SUBROUTINE as an OVERLAY or SEGMENT if the PROGRAM name ( ) statement is used instead of SUBROUTINE name ( ).

6 If a fault checking statement is used in an overlay or segment, SELECT* is used to select the condition. REMOVE* may not be used to remove the condition before sequencing to another overlay or segment, thus leaving an interrupt selection to some meaningless location in later overlays or segments.

7 When an overlay or segment uses BCD input, a record may be lost when sequencing between overlays or segments because of the one-record-ahead buffering scheme in BCD input.

11.1
CALLING SEQUENCE

The FORTRAN calling sequences for overlays and segments are:

```
CALL OVERLAY (n,p,o)

CALL SEGMENT (n,p,o,s)
```

n is the logical unit from which the overlay or segment is to be loaded.

p is the parameter to be passed to the routine.

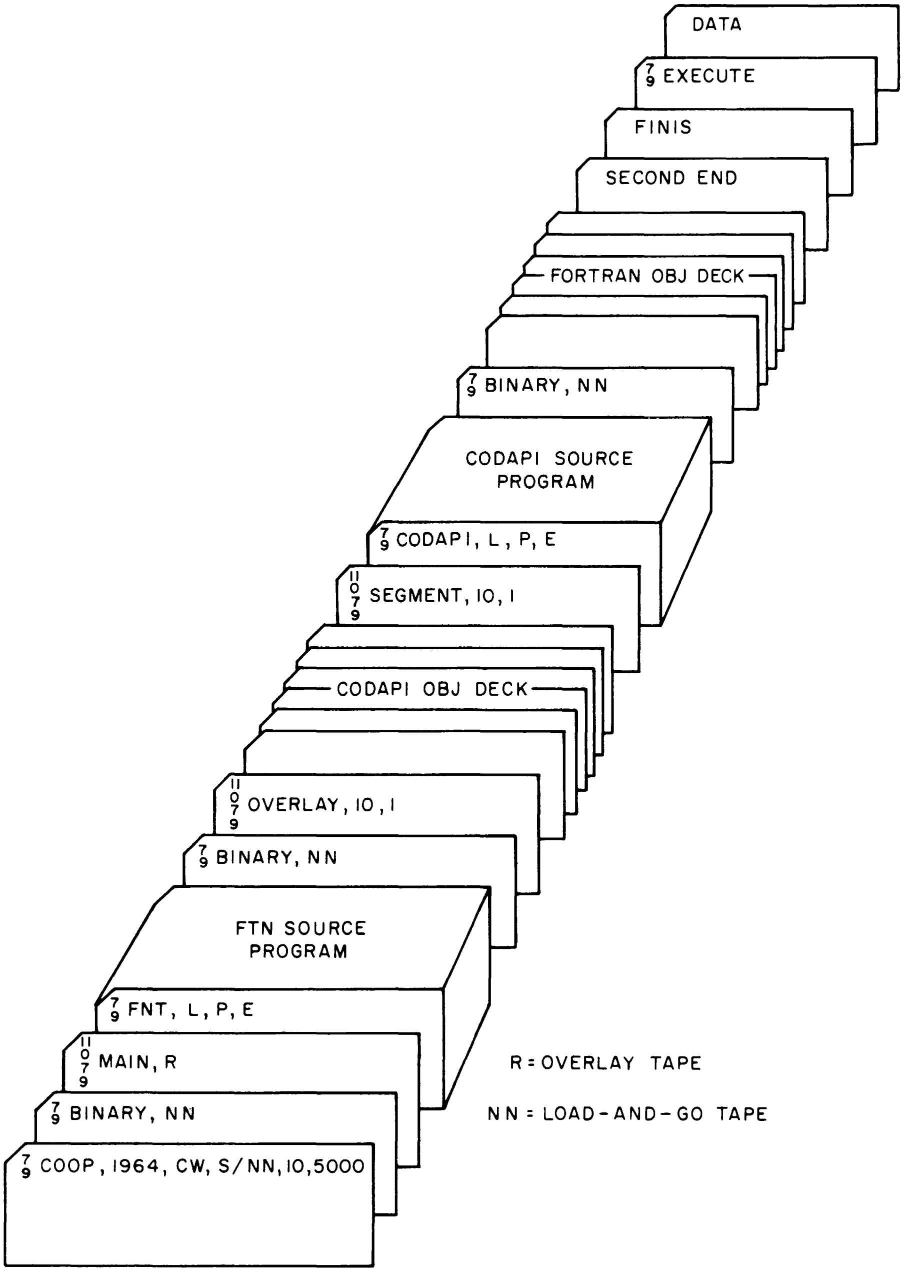
o is the number of the overlay.

s is the number of the segment.

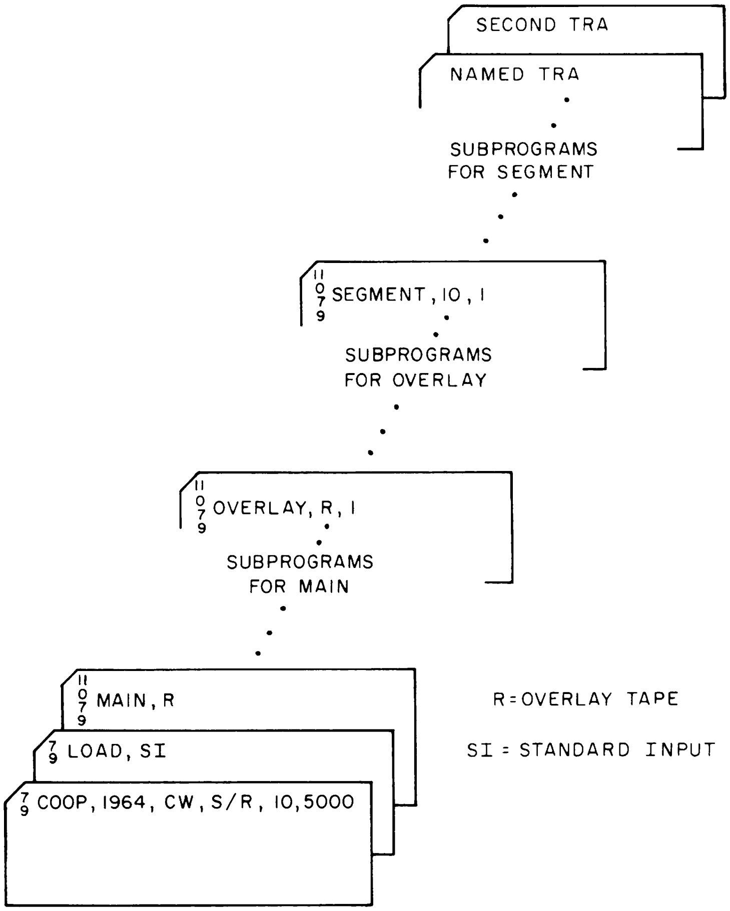
11.2

DECK STRUCTURES

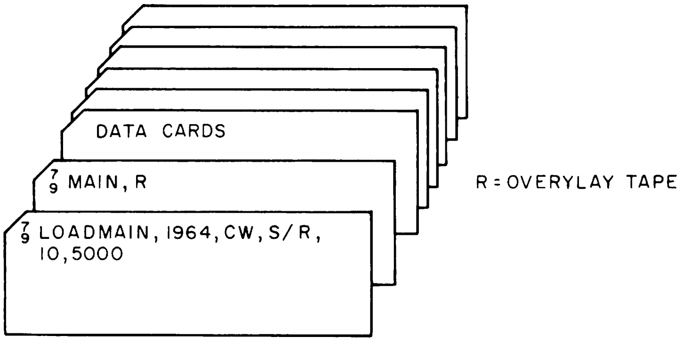
A typical load-and-go job consisting of a main subprogram, an overlay and a segment might be set up according to the accompanying diagram.



If it is desirable to generate an overlay tape for execution at a later time, the control cards would be placed as follows:



At the time the prepared overlay tape is to be executed, the deck would look like this:



APPENDIX SECTION

CHARACTER CODES A

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	11
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

STATEMENTS OF FORTRAN-63

B

			Page
SUBPROGRAM STATEMENTS			
ENTRY POINTS	PROGRAM name	N*	7-1
	PROGRAM name (p_1, ... , p_n)	N	7-1
	SUBROUTINE name	N	7-8
	SUBROUTINE name (p_1, p_2, . . .)	N	7-8
	FUNCTION name (p_1, p_2, . . .)	N	7-2
	ENTRY name	N	7-13
INTER-SUBROUTINE TRANSFER STATEMENTS	EXTERNAL $name_1$, $name_2$, . . .	N	7-5
	CALL name	E	7-8
	CALL name (p_1, ... , p_n)	E	7-8
	RETURN	E	7-12
DATA DECLARATION AND STORAGE ALLOCATION			
TYPE DECLARATIONS	TYPE COMPLEX List	N	4-1
	TYPE DOUBLE List	N	4-1
	TYPE REAL List	N	4-1
	TYPE INTEGER List	N	4-1
	TYPE LOGICAL List	N	4-1
	TYPE name # (w,/b) List # is 5, 6, 7	N	5-2
STORAGE ALLOCATIONS	DIMENSION V_1, V_2, ... , V_n	N	4-2
	COMMON/I_i/ List . . .	N	4-3
	EQUIVALENCE (A,B, ...), (A1,B1, . . .) . . .	N	4-7
	DATA (I_1 = List), (I_2 = List), . . .	N	4-9
ARITHMETIC STATEMENT FUNCTION			
	Function (p_1, ... , p_n) = Expression	E	7-6
SYMBOL MANIPULATION, CONTROL AND I/O			
REPLACEMENT	A = E Arithmetic	E	2-1
	L = E Logical/Relational	E	3-1
	M = E Masking	E	3-6
	A_m = . . . = A_1 = E Multiple	E	3-8
INTRA-PROGRAM TRANSFERS	GO TO n	E	6-2
	GO TO m, (n_1, . . . n_m)	E	6-2
	GO TO (n_1, ... , n_m)i	E	6-2
	GO TO (n_1, ... , n_m),i	E	6-2
	IF (A) n_1,n_2,n_3	E	6-3
	IF (L) n_1,n_2	E	6-3
	IF (SENSE LIGHT i)n_1,n_2	E	6-4
	IF (SENSE SWITCH i)n_1,n_2	E	6-4

---

*N = Non-executable E = Executable

Statements of FORTRAN-63 (Continued)

			Page
	IF DIVIDE $\left\{\begin{matrix}\text{FAULT}\\\text{CHECK}\end{matrix}\right\}$ n_1,n_2	E	6-4
	IF EXPONENT FAULT n_1,n_2	E	6-5
	IF OVERFLOW FAULT n_1,n_2	E	6-5
	IF (EOF, i) n_1,n_2	E	9-11
	IF (IOCHECK, i) n_1,n_2	E	9-11
LOOP CONTROL	DO n i = m_1,m_2,m_3	E	6-5
MISCELLANEOUS PROGRAM CONTROLS	ASSIGN s TO m	E	6-2
	SENSE LIGHT i	E	6-4
	CONTINUE	E	6-8
	PAUSE; PAUSE n	E	6-9
	STOP; STOP n	E	6-9
I/O FORMAT	FORMAT ($spec_1$, $spec_2$, . . .)	N	8-3
I/O CONTROL STATEMENTS	READ n, L	E	9-6
	PRINT n, L	E	9-1
	PUNCH n, L	E	9-2
	READ (i,n) L READ INPUT TAPE i,n,L	E	9-6
	WRITE (i,n) L WRITE OUTPUT TAPE i,n,L	E	9-2
	READ (i) L READ TAPE i,L	E	9-7
	WRITE (i) L WRITE TAPE i,L	E	9-2
	BUFFER IN (i,p) (A,B) BUFFER OUT (i,p) (A,B)	E	9-8
I/O TAPE HANDLING	END FILE i	E	9-10
	REWIND i	E	9-10
	BACKSPACE i	E	9-10
INTERNAL DATA MANIPULATION	ENCODE (c,n,V)L	E	9-14
	DECODE (c,n,V) L	E	9-14
PROGRAM AND SUBROUTINE TERMINATION	END	N/E	6-9, 7-13

LIBRARY FUNCTIONS AND DIAGNOSTICS

C

Diagnostic print-outs will be of the form:

ERROR IN XXXXXXXXXXXXXX A = 0000000000000000 CALL FROM ZZZZZ where

XXXXXXXXXXXXXX = name of routine in which error occurred.

A = contents of the A-register

ZZZZZ = the location from which the routine was called

NAME OF ROUTINE	DEFINITION	PARAMETER MODE	RESULT MODE	TYPE OF ERROR	CONTENTS OF A
ABSF(X)	Absolute Value of X	Real	Real	-	-
XABSF(i)INTF(X)	Absolute Value of i Truncated	Integer	Integer	-	-
INTF(X)	Truncation of Integer X	Real	Real	-	-
MODF(X_1,X_2)	X_1 modulo X_2	Real	Real	Second Argument (Div.) = 0	First Argument
XMODF(i_1,i_2)	i_1 modulo i_2	Integer	Integer	Second Argument (Div.) = 0	First Argument
MAX0F (i_1,i_2, . . .)	Determine Max Argument	Integer	Real	-	-
MAX1F(X_1,X_2, . . .)	Determine Max Argument	Real	Real	-	-
XMAX0F (i_1,i_2, . . .)	Determine Max Argument	Integer	Integer	-	-
XMAX1F(X_1,X_2, . . .)	Determine Max Argument	Real	Integer		
MIN0F (i_1,i_2, . . .)	Determine Min Argument	Integer	Real	-	-
MIN1F(X_1,X_2, . . .)	Determine Min Argument	Real	Real	-	-
XMIN0F (i_1,i_2, . . .)	Determine Min Argument	Integer	Integer	-	-
XMIN1F(X_1,X_2, . . .)	Determine Min Argument	Real	Integer	-	-
SINF(X)	Sine X Radians	Real	Real	$\lvert X \rvert > 2^{36}$	Argument
COSF(X)	Cosine X Radians	Real	Real	$\lvert X \rvert > 2^{36}$	Argument
TANF(X)	Tangent X Radians	Real	Real	$\lvert X \rvert > 2^{36}$	Argument
ASINF(X)	Arcsine X Radians	Real	Real	$\lvert X \rvert > 1$	Argument
ACOSF(X)	Arccosine X Radians	Real	Real	$\lvert X \rvert > 1$	Argument
ATANF(X)	Arctangent X Radians	Real	Real	-	-
TANHF(X)	Hyperbolic Tangent X Radians	Real	Real	-	-
SQRTF(X)	Square Root of X	Real	Real	$X < 0$	Argument
LOGF(X)	Natural Log of X	Real	Real	$X \leq 0$	Argument
EXPF(X)	e to Xth power	Real	Real	$X > 709.0895$	Argument
SIGNF(X_1,X_2)	Sign of X_2 times $\lvert X_1 \rvert$	Real	Real	-	-
XSIGNF(i_1,i_2)	Sign of i_2 times $\lvert i_1 \rvert$	Integer	Integer	-	-

NAME OF ROUTINE	DEFINITION	PARAMETER MODE	RESULT MODE	TYPE OF ERROR	CONTENTS OF A
DIMF(X_1,X_2)	for $X_1 > X_2 : X_1 - X_2$ for $X_1 \leq X_2 : 0$	Real	Real	Overflow Occurred	First Argument
XDIMF(i_1,i_2)	for $i_1 > i_2 : i_1 - i_2$ for $i_1 \leq i_2 : 0$	Integer	Integer	Arithmetic Overflow Occurred	-
CUBERTF(X)	Cube root of X	Real	Real	-	-
FLOATF(i)	Integer to Real Conversion	Integer	Real	-	-
RANF(N)	Generate Random Number	Negative Positive	Real Integer	-	-
XFIXF(X)	Real to Integer Conversion	Real	Integer	$X > 2^{47}-1$	Argument
XINTF	is equivalent to XFIXF				
POWRF(X_1,X_2)	$X_1^{X_2}$	Real, Real	Real	Base < 0 (exp)(ln base) > 709.0895 Base = 0, exp ≤ 0	First argument (exp)(ln base) First argument
ITOJ(I,J)	I^J	Integer, Integer	Integer	Exp > 47 (exp)(ln base) > 709.0895	Second Argument -
XTOI(X,I)	X^I	Real, Integer	Real	Base = 0, exp ≤ 0	First argument
ITOX(I,X)	I^X	Integer, Real	Real		
LENGTHF(i)	Number of words read on logical unit i	Integer	Integer		
DPOWER(Z_1,Z_2)	$Z_1^{Z_2}$	Double, Double	Double		
*DCUBRT(Z)	Double precision cube root of Z	Double	Double		
*DATAN(Z)	Double precision arctangent of Z radians	Double	Double		
*DSIN(Z)	Double precision sine of Z radians	Double	Double		
*DCOS(Z)	Double precision cosine of Z radians	Double	Double		
*DEXP(Z)	Double precision exponential of Z	Double	Double		
*DSQRT(Z)	Double precision square root of Z	Double	Double		
*DLOG(Z)	Double precision natural logarithm of Z	Double	Double		

---

*These functions are not presently on the define tape.

INPUT/OUTPUT DIAGNOSTICS D

ERROR MESSAGE FORMAT

ERROR IN XXXXXXXXXXXXX A = YYYYYYYYYYYYYYYY CALL FROM ZZZZZ

XXXXXXXXXXXXX	identifies the I/O routine in use at the time of the error and also indicates the type of error.
YYYYYYYYYYYYYYYY	the A register will contain: a) Value of p when the terminating error code is MODE b) Number of errors when the terminating error code is DATA c) Logical tape number in all other cases
ZZZZZ	designates the address from which the I/O function was called.

Input/output routines which may give rise to error conditions are:

BCD IN
BCD OUT
BIN IN
BIN OUT
BUFINOUT

Terminating Error Codes	Description
TAPE	Tape number was not defined or was out of range.
FORM	FORTRAN FORMAT specification or parameter list was incorrect.
DATA	Input character indicated by the FORMAT statement is not legal for this type conversion.
WIDTH	BCD record length described by the FORMAT statement is too long for the specified unit.
NO D	Double precision conversion has been requested, but no variables have been declared double precision type. The double precision routines, therefore, are not available in core.
ROOM	More than 16 buffered tapes have been requested at one time.

Terminating Error Codes	Description
SYNC	A discrepancy between the lengths of the physical and logical records on the binary input has been detected.
MODE	$p \neq 0, 1$, or 2 in a BUFINOUT statement.
RECL	A record has been encountered containing 1, or less, word; it has been interpreted as noise.
LIST	The INPUT/OUTPUT list has requested more data than is available in the logical record.
EOF	An end-of-file was encountered before the end of a logical record on an input statement.

When an unrecoverable error occurs while writing tape or punching cards, a non-terminating BCD OUT message will result.

XX T NN

XX	equals	PE	for a parity error
		BE	for a buffer length error
		PB	for buffer length and parity error
		CE	if the on-line card punch fails

T	is the octal number of the tape.
NN	is the octal number of the logical unit.

OPERATIONS AND CALLING SEQUENCES

To understand the following discussions, the programmer must be familiar with CODAP1 instructions and coding procedures. The detailed discussion of calling sequences for standard arithmetic expressions should aid the user in writing additional functions and non-standard type arithmetic subroutines.

A

STANDARD ARITHMETIC EXPRESSIONS

A.1

INSTRUCTION TYPES

During compilation of an expression, the translator generates the following instruction types to execute the operations indicated by the operators.

Instruction Types	Operators
Add operand	+
Subtract operand	-
Multiply operand	*
Divide operand	/
Complement accumulator	-(unary)
Power	**
Load operand Load negative operand Store operand	operand manipulations

Instructions are generated independently of the arithmetic mode and type of operand. The mode of the accumulator and operands as well as the element size are determined from the TYPE declarations or the variable name convention. For standard types (real, integer, double, complex, logical), these are fixed. The appropriate machine order, or a jump to a routine which executes the intended operation then replaces the generated instruction type.

A.2

CALL IDENTIFIER

Load and load complement instructions for all modes and arithmetic involving reals or integers exclusively generate CODAP1 machine instructions. In other words, these operations are performed in-line.

To perform double and complex operations (other than load and load complement) and conversions for mixed mode arithmetic, the compiler generates library routine calls which have the form:

QnQOOmst

n indicates the number of operands to be treated.

n = 0 for operations on the accumulator only.

n = 1 if the operand is a full or multiple word element.

n = 2 for exponentiation; exponentiation is not defined for partial word operands.

n = 3 if the operand is a partial word or byte-sized element.

OO indicates the operation code. The operation is determined by the operator in the expression.

00	Load accumulator with operand
01	Load accumulator with complement of operand
02	Add operand to accumulator
03	Subtract operand from accumulator
04	Multiply accumulator by operand
05	Divide accumulator by operand
06	Complement accumulator
07	Raise operand_1 to the power operand_2
10	Store accumulator in operand

m indicates the mode of the accumulator before store operations and after all other operations.

0	mode is integer
1	mode is real
2	mode is double
3	mode is complex
4	mode is logical
5	mode is non-standard
6	mode is non-standard
7	mode is non-standard

s indicates the mode of the operand. The values of s are the same as those defined for m.

t indicates the mode of the exponent. It appears only with identifiers of the form Q2Q07mst; for other QnQ identifiers, it is always 0. Exponentiation involving a partial word operand is not permitted, except where the exponent is an integer constant 1-8.

Example:

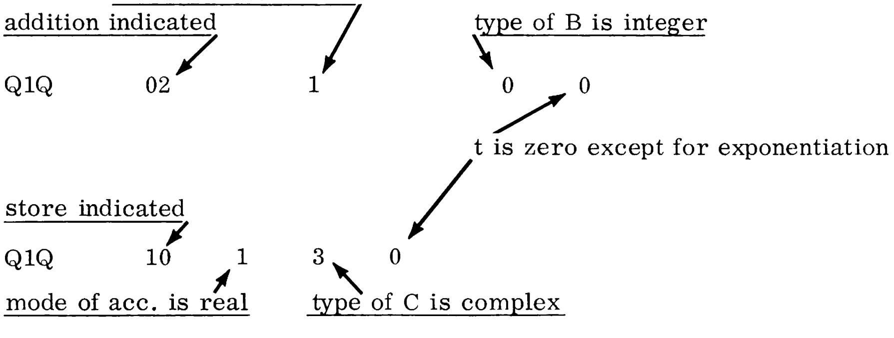
```
TYPE REAL A
TYPE INTEGER B
TYPE COMPLEX C
C = (A + B)
```

Translator Instructions	Conversions	Call Identifier
Load A	none	none
Add B	integer to real	Q1Q02100
Store C	real to complex	Q1Q10130

The resulting CODAP1 object code:

		Interpretation
	LDA A	transmit contents of location A to accumulator
+	CALL Q1Q02100	go to subroutine, convert B to real and add to accumulator
	00 B	
+	CALL Q1Q10130 00 C	go to subroutine, convert accumulator to complex and store accumulator in C.

Breakdown of the QnQ identifiers used in the example:



A.3
CALLING SEQUENCES

Standard groups of CODAP1 instructions are generated when jumps are made to QnQ subroutines, library functions, and subprograms.

A.3.1
MIXED-MODE ARITHMETIC, DOUBLE AND COMPLEX OPERATIONS

If the operand is a parameter in a subroutine or function, it appears in the object code as **.

Q0Q SUBROUTINES

For operation 06, complement accumulator, the following code is generated:

```
L    CALL   Q0Q06mst
L+1 Return
```

Q1Q SUBROUTINES

For full word operand (1 to 7 words per operand) and all operations except 06 and 07, the code generated is:

```
L     CALL    Q1QOOmst
         0   b   operand + constant addend
L+1  Return
```

b is an index designator; the content of b is an indexing quantity (index function) reflecting variable subscripts on the operand.

constant addend is a bias on the base address to balance a portion of the index function contained in b, or simply a position relative to the base array address of a variable with constant subscripts. To calculate the constant addend and (b) for element A $(\ell_i * i \pm c_i, \ell_j * j \pm c_j, \ell_k * k \pm c_k)$ in array A(I,J,K) the following formula is used.

Base Address Constant Addend Index Function

$$\text{Locn } A + (-\ell \pm c_i + I * (-\ell \pm c_j + J * (-\ell \pm c_k)))*f + (\ell_i * i + I * (\ell_j * j + J * (\ell_k * k)))*f$$

$\ell_i, \ell_j, \ell_k, c_i, c_j, c_k$ are unsigned integer constants
f is the element length (1-7 words)

The effective operand address is (b) + operand + constant addend.
b, (b) and/or the constant addend may be 0.

Q2Q SUBROUTINES

For operation 07, exponentiation, the following code is generated:

```
+       SLJ          *+1
         0      b1   operand1 + constant addend1
L       CALL         Q2Q07mst
         0      b2   operand2 + constant addend2
L+1     Return
```

b_1, b_2, etc. are defined in Q1Q calling sequence.

Q3Q SUBROUTINE

For partial word operand, logical, the calling sequence is:

```
+       SLJ          *+1
        n       b    constant addend
L       CALL         Q3QOOmst
        POF     0    operand
L+1     Return
```

n	is the element length in bits
POF	is the parameter offset which appears in the object code as 00. An offset is the number of bits between the left end of the word and the logical bit. The parameter offset is passed along with the operand address when the operand is a parameter in a subroutine call. During execution, it is transmitted with the parameter to all Q3Q calls within the subroutine. If there is no offset or if the operand is not a parameter in a subroutine call, the POF will be zero.

For logical arithmetic, the effective operand address is computed as follows by an object time routine:

a.d = ((n*((b)+ca))+POF)/p

a = first word address (FWA) addend (quotient)

d = actual offset (remainder)

n = element length in bits

(b) = content of index register

ca = constant addend

POF = parameter offset

p = packing number (32 bits per word for logical;
48 bits per word for byte)

The effective operand is the n bits of word FWA + a, d bits from left.

For more information and an example of the CODAP1 Q3Q Calling Sequence for non-standard byte operations, see page G-11.

A.3.2
SUBPROGRAMS

The subprograms (function or subroutine) are called by the following sequence. The parameters will appear as ** in the object code if they are parameters in other subroutines or functions.

	RTJ	SUBNME	
+	0	Parameter 1	address of actual parameters
-	0	Parameter 2	
+	0	Parameter 3	
-	0	Parameter 4	
	⋮	⋮	
+	Return		

or more explicitly

		RTJ	SUBNME	
		(offset)		
.Z#.	+	0	base address + FWA addend	if actual parameter specifies a partial word element
	-	0	effective address	if actual parameter specifies a multi-word element

When the call for a subprogram with partial-word actual parameters is generated, the offset is calculated by a special library routine Q9QEVALB. The offset is made available to the subprogram at execution time by storing it with the parameter relative to the word tagged .Z#. See example III for the use of Q9QEVALB and the call to subprogram with parameter offsets, page G-19.

Examples:

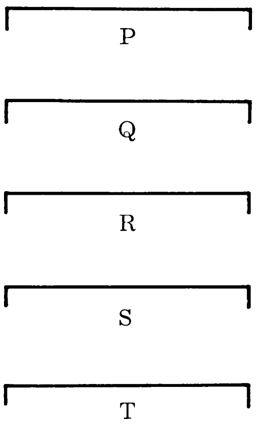
1) Function Subprogram Reference

```
Z=QUAINT  (P,Q,R,S,T)
```

results in call

```
        RTJ     QUAINT
+        0        P   \
-        0        Q    |
+        0        R    >  non-subscripted multi-word elements
-        0        S    |
+        0        T   /
+       Return
```

in memory



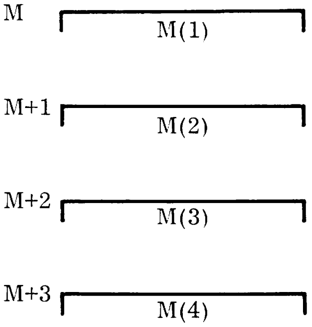
2) Subroutine Subprogram

```
CALL    SAM      (M,M(3), M(4) )
        M is one word per element
```

results in call

```
        RTJ     SAM
+        0      M         M is address of operand
-        0      M+2       effective address is the third word
+        0      M+3       effective address is the fourth word
+       Return
```

in memory



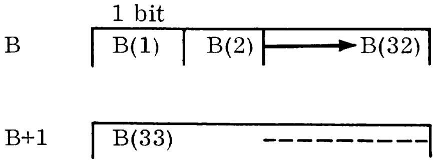
CALL SAM (B, B(2), B(33))

B is an array of logical elements

results in call

```
            RTJ     SAM
.Z#.  +      0      B        B(1) element is leftmost character in
                             first word

      -      0      **       B(2) element has offset of 1 and is in
            (1)     (B)      first word


      +      0      **       B(33) element is leftmost bit in second
                    (B+1)    word

                             The values in the parentheses indicate
                             the contents of the word at object time.
```

in memory



A.3.3 LIBRARY FUNCTIONS

Library functions have two entry points as they may be called by value or by name. Some are also called for expression evaluation and these are named with the conventions for mixed mode arithmetic. The instruction word in the parentheses will be present in function calls with two parameters.

The call by value generates the following sequence; the actual parameter is passed to the A or Q register or both.

```
        LDA     Parameter
        LDQ     Parameter
        RTJ     Function

    +   Return
```

The call by name generates the following sequence; the address of the parameter is stored in the computer word following the RTJ instruction.

```
        RTJ     Q8Qfunction

    +    0      Parameter
   (-    0      Parameter)
    +   Return
```

The typical library function entry points are then

```
Q8Q function     NOP     **           call by name
                 RTJ     Q8QLOADA
Function         SLJ     **           call by value
                  .       .
                  .       .
                  .       .
```

The call by name transfers to the special routine Q8QLOADA which analyzes the call by name and makes it a call by value; the routine is then executed as if it had been called by value.

The following are examples of FORTRAN coding that give rise to the different means of calling the library routines

```
                  FORTRAN  generates    CODAP 1

Call by Value:    X=SINF(X)             LDA  X

                                        RTJ  SINF

Call by Name:     EXTERNAL SINF                RTJ    PHI

                  Z = PHI (X, SINF)  calling  +  0      X
                                     program     0      Q8QSINF

                  FUNCTION PHI (P,Q)          FP00001. RTJ   **
                  PHI = Q (P)                                (Q8QSINF)
                                     function
                  END                PHI      FP00002. 0     **
                                                             (X)
```

B

NON-STANDARD ARITHMETIC EXPRESSIONS

To implement a non-standard type arithmetic, it is necessary to write a set of routines which have the entry points generated by the compiler as externals (EXT) when an expression is evaluated. These routines must define the expressions which contain operands of different type (conversion routines for mixed mode) and define the operations. The mode of the accumulator and operands and the element size are defined by the TYPE-other declaration. The form of the call identifiers and calling sequences are the same for non-standard arithmetic as for standard.

These routines can be written in any compiler or assembly language. Routines handling byte arithmetic are usually written in an assembly language to facilitate offset and constant addend manipulations. All non-standard operations must be performed in user-provided routines. If the required routine for an operation is not available, a load time diagnostic occurs.

B.1

CALLING SEQUENCES

B.1.1

ALL ARITHMETIC OPERATIONS AND MIXED-MODE CONVERSIONS

QnQ SUBROUTINES

For multi-word elements, same as standard.

The programmer must supply the routines for the Q0Q, Q1Q, Q2Q call identifiers.

Example:

```
          PROGRAM OTHER5
          TYPE BYTE5(/8) A,B,C,D
          TYPE QUAD6(4) AX,BX,CX,DX
          DIMENSION D(20),DX(10)
1         A=B+C                          .4      CALL     Q1Q00660
2         AX=BX+CX                                0       BX
3         A=B*C                          +       CALL     Q1Q04660
4         AX=BX*CX --------------------->         0       CX
5         A=D(5)+D(8)----\               +       CALL     Q1Q10660
6         AX=DX(3)+DX(4)  \                       0       AX
7         I=A+B            \             .5      SLJ      *+1
8         C=I*A             \                    10       +4
9         J=I/C              \                   CALL     Q3Q00550
10        A=I-J               \                   0       D
11        R=A+B                \         +       SLJ      *+1
12        S=R+A                 \                10       +7
13        A=R+S                  ------>         CALL     Q3Q02550
14        C=R+A                                   0       D
15        IX=AX+BX                       +       SLJ      *+1
16        RX=AX-BX                               10       0
17        CX=AX+IX                               CALL     Q3Q10550
18        DX(3)=IX+RX                             0       A
19        A=I+R
20        D(5)=I+R
          END
```

Q3Q SUBROUTINES

For byte arithmetic, same as for logical arithmetic.

The offset for a byte is the number of bits between the left end of the word and the leftmost bit of the byte element.

The programmer must include instructions in his Q3Q routine to compute the effective operand address -

$$a.d = ((n*((b)+ca))+POF)/p$$

and to locate the effective operand.

The packing number, p, for bytes is 48 bits per word.

Example:

```
FORTRAN                     CODAP Calling
                            Sequences
PROGRAM OFFSET

DIMENSION A(20)

TYPE OTHER5 (/8)A
      .
      .                           RTJ    SAM
      .
CALL SAM (A(3) )            .Z#.  0      **        The offset is calculated
      .                           (1)    (1)       by the Q9QEVALB routine
      .                           (20)   (A)       and stored with the
      .                                            parameter address at
END                                                location .Z#.

SUBROUTINE SAM(B)

DIMENSION  B(15)
                             +    SLJ    *+1
TYPE OTHER5  (/8)  B
                                  10  (23) -16
I = 23                                             This Q3Q00550 routine
                                                   must compute the
C = B(I-15)
      .                      L    CALL   Q3Q00550  effective operand
      .                                            address; it may call
      .                            0          **   Q9QEVALB to do this.
END                               (20)    (B+1)
```

Calling sequence for Q9QEVALB:

```
          ENQ                            byte size

          ENA          b                 CA

   +      CALL                           Q9QEVALB

          POF          0                 operand

            A  upper half word
          ST( )                          .Z#.
            Q  lower half word
```

b	is the index function
CA	is the constant addend
POF	is the parameter offset

Calculations performed in example:

1) for constant addend and index function

Locn B - (1+15) + (8+15)
Locn B - (16) + (23)

ca = -16
(b) = 23

2) effective operand address

a.d = ((8*(23+(-16))+16)/48

a = 1 - FWA addend
d = 24 - actual offset

In memory

B	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
			B(1)	B(2)	B(3)	B(4)
B+1	A(7)	A(8)	A(9)	A(10)	A(11)	A(12)
	B(5)	B(6)	B(7)	B(8)	B(9)	B(10)
B+2	A(13)	A(14)	A(15)	A(16)	A(17)	A(18)
	B(11)	B(12)	B(13)	B(14)	B(15)	
B+3	A(19)	A(20)				

B.1.2 SUBPROGRAM

The calling sequence is the same for non-standard parameters as for standard parameters.

Example:

DIMENSION B(12)

TYPE OTHER6 (/8) B

CALL SAM (B, B(2), B (11))

results in call

		RTJ	SAM	
.Z#.	+	0	B	The first element of B array is the left-most character of the first word.
	-	0 (10)	** (B)	The second element of B array is offset from the left 8 bits (octal 10) but is still in the first-word.
	+	0 (40)	** (B+1)	The eleventh element of B array is in the second word and is offset 32 bits from the left.
				The values in the parentheses indicate the contents of the word at object time.

in memory

8 bits

B	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)
B+1	B(7)	B(8)	B(9)	B(10)	B(11)	B(12)

B.2 EXAMPLES

I. Polish String - Byte Arithmetic

This subroutine translates a fully parenthesized arithmetic expression into a Lukasiewicz parenthesis-free notation. This example shows a CODAP1 routine with entry points for each call identifier.

```
         PROGRAM STRING
         TYPE INTEGER S,P,T
         DIMENSION S(10),P(10),T(10)
         COMMON I
         PRINT 500
500      FORMAT(115H1 DEMONSTRATION OF A ROUTINE TO CONVERT FULLY PARENTHES
   *     IZED ARITHMETIC STRINGS INTO PARENTHESIS IS FREE POLISH STRINGS   )
         I=1
1        READ 100,S
100      FORMAT(10A8)
         IF(S(1).EQ.8HFINISH  )4,2
2        DO 3 J=1,10
3        T(J)=P(J)=8H
         CALL POLISH(S,P,T,80)
         CALL PRESS(P,80)
         PRINT 300,S,P
300      FORMAT(17H0 INPUT STRING =  10A8/17H0POLISH STRING =  10A8)
         GO TO 1
4        PRINT 400
400      FORMAT(7H0FINISH)
         END

         SUBROUTINE POLISH(S,P,T,N)
         TYPE BYTE5 (/6) S,T,P
         DIMENSION S(N),P(N),T(N)
         COMMON I
         K=1  $  I=N  $  J=N
1        IF(S(J).EQ.1R) )8,2
2        IF(S(J).EQ.1R+.OR.S(J).EQ.1R-.OR.S(J).EQ.1R*.OR.S(J).EQ.1R/)3,4
3        T(K)=S(J) $ K=K+1 $ GO TO 10
4        IF(S(J).EQ.1R()5,6
5        P(I)=T(K-1) $ K=K-1 $ GO TO 7
6        P(I)=S(J)
7        I=I-1
8        IF(J.EQ.1)9,10
9        RETURN
10       J=J-1 $ GO TO 1
         END

         SUBROUTINE PRESS(P,N)
         TYPE BYTE 5 (/6) P
         DIMENSION P(N)
         COMMON I
         K=1$I=I+1
         DO 1 J=I,N
         P(K)=P(J)
1        K=K+1
         DO 2 J=K,N
2        P(J)=(1R )
         END
```

```
          IDENT    BYTES6
          ENTRY    Q3Q00550            ENTRY TO LOAD SIX BIT BYTES
Q3Q00550  SLJ      **
          SIU    1 C
          LIU    1 Q3Q00550
          RTJ      P                   COMPUTE ADDRESS OF OPERAND
          SAU      *+1                 A=OFFSET
          LDA    7 M                   (M)=ADDRESS
          LLS      **
          ENA      0
          LLS      6                   SHIFT IN BYTE
          SLJ      Q3Q00550            RETURN
Q3        SIU    1 C
          LIU    1 Q3Q10550
          ENQ      -0
          LRS      6
          STQ      T                   SAVE BYTE WITH MASK
          RTJ      P                   COMPUTE ADDRESS OF RESULTANT
          SAL      *+1                 A=OFFSET
          LDA      T                   (M)=ADDRESS
          LDQ      =077777777777777    ALL BUT HIGH ORDER CHARACTER
          LRS      **                  POSITION MASK AND BYTE
          SSU    7 M                   MASK ALL BUT NEW 6 BITS FROM
          REMARK                       STORAGE
          STA    7 M                   RESTORE RESULTANT
          ENTRY    Q3Q10550            ENTRY TO STORE SIX BIT BYTES
Q3Q10550  SLJ      **
          SLJ      Q3
P         SLJ      **                  COMPUTE EFFECTIVE ADDRESS OF
          REMARK                       OPERAND AT ((B1)-2 AND (B1)-1)
          LDQ      =0-777777
          LDA    1 -2                  WORD CONTAINS B CONSTANT ADDEND
          REMARK                       IN LOWER ADDRESS
          SSU      C
          STA      C
          LDA    1 -1                  WORD CONTAINS OFFSET 0 BASE ADDRESS
          REMARK                       IN LOWER ADDRESS
          SAL      M
          LRS      24
          ENA      0                   COMPUTE
          LLS      6                   (OFFSET/6+B+CONSTANT ADDEND)/8
          ENQ      0
          DVI      =6
C         ENI    1 **
          INA
          LRS      3
          RAD      M                   =ADDITIVE TO BASE ADDRESS
          ENA      0
          LLS      4                   REMAINDER*6=BYTE OFFSET
          SAU      *+1
          LLS      1
```

```
         LLS        1
         INA        **
         SLJ        P
         ENTRY      Q1Q03500           ENTRY TO SUBTRACT INTEGER FROM
         REMARK                        SIX BIT BYTE IN A
Q1Q03500 SLJ        **
         SAL        Q1
         LDA        *-1
         ALS        24
         INA        -1
         SAU        Q1                 ADDRESS OF  B BASE ADDRESS+CONSTANT
         REMARK                        ADDEND
Q1       LAC     7  **                 COMPLEMENTED OPERAND
         INA        **                 ADD A
         SLJ        Q1Q03500           RETURN
M        OCT        0
T        BSS        1
         END
```

```
PRINTED OUTPUT

DEMONSTRATION OF A ROUTINE TO CONVERT FULLY PARENTHESIZED ARITHMETIC STRINGS
INTO PARENTHESIS FREE POLISH STRINGS
 INPUT STRING = (((A+B)-C)*D)
POLISH STRING = *-+ABCD
 INPUT STRING = (A+(B-(C*D)))
POLISH STRING = +A-B*CD
 INPUT STRING = (((A+B)-(C*D))/E)
POLISH STRING = /-+AB*CDE
 INPUT STRING = (((A+(B-C))*((D/E)+F))-G)
POLISH STRING = -*+A-BC+/DEFG
 INPUT STRING = ((A+B)*(C-D))
POLISH STRING = *+AB-CD
FINISH
```

II. Double Precision Complex - Multi-word Elements

These routines were written to handle double precision complex arithmetic which would extend computational precision to four computer words.

This example shows two variations of FORTRAN routines. The first has entry points for each operation; the second has a separate subroutine for each operation.

```
      SUBROUTINE Q1Q00550(AD)
C                 FTN63BA08          02APD                      L/03/20
C     TYPE 5 - DOUBLE PRECISION COMPLEX ARITHMETIC PACKAGE
C     NOTE THAT ACCUM(1) SHOULD ALWAYS BE INVOLVED IN THE OPERATION JUST
C     PREVIOUS TO RETURN TO INSURE THAT IFS ARE CONSISTENTLY TESTING THE
C     MOST SIGNIFICANT PORTION OF THE REAL PART OF THE VARIABLE.
      DIMENSION ACCUMD(2),AD(2)
      TYPE DOUBLE ACCUMD,AD,B,C
      COMMON/DPCMPLXC/ACCUMD
C
C     LOAD ACCUMULATOR
      ACCUMD(2) = AD(2)
      ACCUMD(1) = AD(1)
      RETURN
C
C     LOAD ACCUMULATOR COMPLEMENT
      ENTRY Q1Q01550
      ACCUMD(2) = -AD(2)
      ACCUMD(1) = -AD(1)
      RETURN
C
C     ADD OPERAND TO ACCUMULATOR
      ENTRY Q1Q02550
      ACCUMD(2) = ACCUMD(2) + AD(2)
      ACCUMD(1) = ACCUMD(1) + AD(1)
      RETURN
C
C     MULTIPLY ACCUMULATOR BY OPERAND
      ENTRY Q1Q04550
      B = ACCUMD(1)*AD(1) - ACCUMD(2)*AD(2)
      ACCUMD(2) = ACCUMD(2)*AD(1) + ACCUMD(1)*AD(2)
      ACCUMD(1) = B
      RETURN

      SUBROUTINE Q1Q00550  (A)
C     LDA-COMPLEX DOUBLE PRECISION-TYPE 5
      DIMENSION A(4)
      COMMON /DPCMPLXC/ACCUM
      ACCUM(1) = A(1)
      ACCUM(2) = A(2)
      ACCUM(3) = A(3)
      ACCUM(4) = A(4)  $  RETURN  $  END
      SUBROUTINE Q1Q01550  (A)
C     LAC-DOUBLE PRECISION COMPLEX
      COMMON/DPCMPLXC/ACCUM
      DO 5 I=1,4
5     ACCUM(I)=-A(I)
      RETURN
      END
```

```
       SUBROUTINE Q1Q02550  (A)
C      ADD-DOUBLE PRECISION COMPLEX-TYPE 5
       COMMON/DPCMPLXC/ACCUM
       TYPE DOUBLE ACCUM,A
       DIMENSION ACCUM(2),A(2)
       ACCUM(1) = ACCUM(1) + A(1)
       ACCUM(2) = ACCUM(2) + A(2)
       RETURN
       END
       SUBROUTINE Q1Q04550  (A)
C      MULTIPLY-DOUBLE PRECISION COMPLEX-TYPE 5
       COMMON/DPCMPLXC/ACCUM
       TYPE DOUBLE ACCUM,A,B
       DIMENSION ACCUM(2),A(2)
       B = ACCUM(1)*A(1) - ACCUM(2)* A(2)
       ACCUM(2) = ACCUM(2)*A(1) + ACCUM(1)*A(2)
       ACCUM(1) = B
       RETURN
       END
```

III. Q9QEVALB Routine

This example shows the CODAP calling sequence for the Q9QEVALB routine to compute parameter offsets.

```
       PROGRAM OTHER5B
COMMENT THIS PROGRAM USES TYPE OTHER VARIABLES IN SUBROUTINE AND
C            FUNCTION CALLS
1      TYPE OTHER5(/3) A,B,SUM
2      TYPE OTHER6(/8) C,D,SUZY
3      TYPE OTHER7(3) E,F
4      DIMENSION A(20),B(40),C(10),D(12),E(10),F(12)
6      EXTERNAL SUZY
5      SUM(X,Y)=X+Y
7      CALL SUZY(D,D(2),D(11))
8      CALL SUZY(A(5),C(2),E(3))
9      CALL NICK(E(6),F(10),SUZY(D,D))
10     CALL NICK(SUM(A,B),A,B)
11     A=MAX(A(2),B(19))
12     B=MAX(B(24),D(5))
13     C=MAX(SUZY(A,B),SUM(A,B))
       END
```

```
.8          ENQ       +3          Number of bits in the element A.
            ENA       +4          Constant addend to base.
+           CALL      Q9QEVALB    Routine calculates the parameter
             0        A           offset and stores it with A in the
            STA       .Z00002.    upper portion of .Z00002.
+           ENQ       +8
            ENA       +1
+           CALL      Q9QEVALB
             0        C
            STQ       .Z00002.    Offset and parameter C is stored in
            RTJ       SUZY        the lower portion of .Z00002.
.Z00002.     0        **
-            0        **
+            0        E+6         Parameter is a multi-word element.

.12         ENQ       +3          Offset calculations and calling
            ENA       +23         sequence are the same for function
+           CALL      Q9QEVALB    subprograms as for subroutines.
             0        B
            STA       .Z00004.
+           ENQ       +8
            ENA       +4
+           CALL      Q9QEVALB
             0        D
            STQ       .Z00004.
            RTJ       MAX
.Z00004.     0        **
-            0        **
+           SLJ       *+1
             3        0
            CALL      Q3Q10050
             0        B
```

IV. Logical and Relational Expressions With Non-Standard Variables

Logical operations are compiled as arithmetic load, test, and store routines; relational operations, as load-load complement, subtract-add, and store routines.

```
        PROGRAM OTHER9A
COMMENT THIS PROGRAM USES TYPE OTHER VARIABLES IN LOGICAL
C       STATEMENTS
        TYPE OTHER5(/3) A.B.C
2       TYPE OTHER6(4) D.E.F
3       TYPE LOGICAL L.M.N
4       L=A.AND.B
5       L=M.OR.A
6       L=.NOT.B
7       L=((A.AND.C).OR.B).AND.D
        N=X.OR.C
9       M=D.AND.E
10      M=N.OR.F
11      M=.NOT.D
12      M=((D.AND.F).OR.E).AND.D
13      N=Z.AND.F
14      L=A.GT.B
15      L=C.LE.A
16      L=B.EQ.C
17      M=A.GE.C.AND.B.EQ.C
18      M=D.LT.E
19      M=E.EQ.F
10      M=F.GT.D
21      N=.NOT.B.GE.C
22      N=.NOT.E.EQ.D
        END
```

```
.9              CALL        Q1Q00660        Routine to load D
                 0          D
                AJP      1  IF00023.
                SLJ         IF00022.        Test D for true or false
IF00023.        CALL        Q1Q00660        Load E
                 0          E
                AJP      1  IF00021.
                SLJ         IF00022.        Test E for true or false
IF00021.        ENA         +1
                SLJ         IF00021.+2
IF00022.        ENA         0
+               SLJ         *+1
                 1          0
                CALL        Q3Q10640        Store 1 or 0 in M
                 0          M
```

```
.14             SLJ         *+1
                 3          0
                CALL        Q3Q01550       Load complement to A
                 0          A
+               SLJ         *+1
                 3          0
                CALL        Q3Q02550       Add B
                 0          B
                AJP     3    IF00041.
                SLJ          IF00042.      Test result
IF00041.        ENA         +1
                SLJ          IF00041.+2
IF00042.        ENA         0
+               SLJ         *+1
                 1          0
                CALL        Q3Q10540       Store 1 or 0 in L
                 0          L
```

COMPILATION DIAGNOSTICS F

Diagnostics prepared by the compiler during compilation are output with the program listing and immediately follow the source program.

FORTRAN-63 diagnostics give the error message, the statement number in which the error occurred or the number of statements beyond the last numbered statement, and the error code.

Examples:

```
FORTRAN-63 DIAGNOSTIC RESULTS
ERROR TYPE G001 DETECTED AT 3 STATEMENTS BEYOND STATEMENT NO. 3
PARENTHESIS USAGE OR DO LOGIC OR TYPE IDENTIFIER IS ILLEGAL IN I/O DATA LIST

ERROR TYPE S021 DETECTED AT STATEMENT NO. 10
A DO LOOP WHICH TERMINATES AT THIS STATEMENT INCLUDES A DO
LOOP WHICH HAS NOT YET BEEN TERMINATED.
```

DIAGNOSTICS

POSSIBLE MACHINE OR COMPILER ERRORS

K467	AN UNIDENTIFIED ERROR HAS OCCURRED. IT MAY BE DUE TO A MACHINE ERROR. RESUBMIT THIS PROBLEM. IF ERROR PERSISTS, SEND SOURCE LISTING TO CONTROL DATA CORP. 3330 HILLVIEW PALO ALTO, CALIFORNIA
B145	COMPILER OR MACHINE ERROR, COMMON IDENT NOT IN DIMENLIS
B150	MACHINE OR TABLE ERROR,VARIABLE NOT IN DIMENLIS.
B205	PROCESS PI ERROR IN HANDLING COMMON EXPRESSIONS.
H003	POSSIBLE MACHINE ERROR. CONFLICT IN DATA IN FUNLIST AND DIMENLIS
H021	POSSIBLE MACHINE ERROR. ARITHMETIC FAULT TYPE NOT RECOGNIZED.
H022	POSSIBLE MACHINE ERROR. MACHINE CONDITION TEST NOT RECOGNIZED.
H107	POSSIBLE MACHINE ERROR. LOGICAL OPERATOR NOT RECOGNIZED
H110	POSSIBLE MACHINE ERROR IN EVALUATING LOGICAL EXPRESSION.
W001	TYPE OTHER OPERAND DOES NOT APPEAR IN DEVARLIS. POSSIBLE MACHINE ERROR.

FATAL ERRORS - ERRORS WHICH TERMINATE COMPILATION

S050	NO END CARD APPEARS IN THIS PROGRAM
B004	NAME NOT STARTING WITH ALPHABETIC CHARACTER.
B005	DUPLICATE VARIABLE NAME IN DIMENSION STATEMENT.
B006	NO LEFT PARENS AFTER VARIABLE NAME
B007	VARIABLE DIMENSION IDENTIFIER NOT IN PARAMETER LIST
B010	MORE THAN 3 DIMENSIONS IN DECLARATION OF ARRAY.
B011	NO RIGHT PARENTHESIS DELIMETER IN SUBSCRIPT DECLARATION.
B144	COMPILER ERROR, TABLE FULL
B146	COMPILER COMMON OR BLOCK TABLE EXCEEDED.
B147	COMPILER ERROR-EQUIVALENCE TABLE EXCEEDED.
K001	SOURCE PROGRAM EXCEEDS CAPACITY OF FORTRAN WITHOUT AN INTERMEDIATE TAPE RE-COMPILE, AND ASSIGN A SCRATCH TAPE.
C050	NUMBER OF FUNCTIONS EXCEED COMPILER LIMIT
C052	NUMBER OF IDENTIFIERS EXCEEDS COMPILER LIMIT
W002	ERASABLE STORAGE REQUIRED IS TOO LARGE.

DESTRUCTIVE ERRORS - ERRORS WHICH PREVENT EXECUTION

S002	A PREVIOUS DO TERMINATES ON THIS DO STATEMENT
S003	A RUNNING INDEX USED IN THIS STATEMENT HAS BEEN USED PREVIOUSLY IN THIS NEST
S004	THE NESTING CAPACITY OF THE COMPILER HAS BEEN EXCEEDED
S005	THE CONSTANT PARAMETERS OF A DO OR DO-IMPLYING LOOP CANNOT EXCEED 32767
S006	THE PARAMETERS OF A DO OR DO-IMPLYING LOOP MUST BE UNSIGNED INTEGER CONSTANTS OR SIMPLE INTEGER VARIABLES.
S007	THE INITIAL VALUE OF A DO OR DO-IMPLYING LOOP MUST NOT EXCEED THE UPPER BOUND IF BOTH ARE CONSTANT
S010	THE RUNNING SUBSCRIPT IN A DO OR DO-IMPLYING LOOP MUST BE A SIMPLE INTEGER VARIABLE
S014	ALL DECLARATIVE STATEMENTS MUST PRECEED THE FIRST EXECUTABLE STATEMENT
S015	THE NUMBER OF INDEX VARIABLES EXCEEDS THE CAPACITY OF THE COMPILER
S017	A DO LOOP TERMINATES AT THIS STATEMENT
S020	A DO LOOP MAY NOT TERMINATE AT AN END STATEMENT
S021	A DO LOOP WHICH TERMINATES AT THIS STATEMENT INCLUDES AN UNTERMINATED DO
S022	THIS STATEMENT DOES NOT FOLLOW A DO WHICH IT TERMINATES
S023	STATEMENTS LABELS MUST BE BETWEEN 1 AND 99999
S024	NON-STANDARD INDEXING IS NOT PERMITTED IN DO STATEMENTS
S025	THE TERMINAL LABEL OF A DO MUST BE AN INTEGER CONSTANT
S026	THIS ENTRY NAME HAS BEEN USED PREVIOUSLY
S027	THE MAXIMUM PERMISSABLE NUMBER OF ENTRY STATEMENTS IS 20
S031	IF THIS IS AN ARITHMETIC STATEMENT, IT HAS NO LEFT HAND SIDE
S032	THE OBJECT OF AN ASSIGN OR ASSIGNED GO TO MUST BE A SIMPLE INTEGER VARIABLE
S036	THE SUBROUTINE NAME IS NOT LEGITIMATE
S037	THE PARAMETER STRING IS NOT WELL-FORMED
S040	THE ASSIGNED STATEMENT LABEL IS NOT AN INTEGER
S042	SUBPROGRAM OR VARIABLE NAME USED AS ENTRY.

S051	THE ENTRY STATEMENT MAY NOT OCCUR INSIDE A DO LOOP
S053	THE INCREMENT IN A DO OR DO-IMPLYING LOOP MUST NOT BE ZERO.
S501	A REAL CONSTANT IN THIS STATEMENT EXCEEDS 2**1023-2**987
S502	ONLY THE DIGITS 01234567 MAY APPEAR IN AN OCTAL NUMBER
S503	AN OCTAL NUMBER MAY HAVE AT MOST 16 DIGITS
S504	ONLY ONE DECIMAL POINT MAY APPEAR IN A CONSTANT
S505	AN ILLEGAL CHARACTER APPEARS IN A NUMERIC FIELD IN THIS STATEMENT
S506	AN ILLEGAL CHARACTER APPEARS IN AN EXPONENT FIELD IN THIS STATEMENT
S507	EXPONENTS ARE LIMITED IN MAGNITUDE TO 309
S510	INTEGERS MAY NOT EXCEED 2**47-1 IN THIS MACHINE
S777	MORE THAN 100 ERRORS WERE DETECTED BY THE COMPILER THE FIRST 100 ARE RECORDED ABOVE.
B002	IMPROPER FORMAT OF PROGRAM STATEMENT.
B003	IMPROPER SUBROUTINE OR FUNCTION STATEMENT TERMINATION OR PARAMETER ERROR
B012	VARIABLE DIMENSIONED ARRAY USED IN COMMON.
B015	NO SLASH (/) SEPARATOR IN BLOCK DESIGNATION.
B016	UNDEFINED SEPARATOR IN COMMON STATEMENT.
B017	NON-CONSTANT SUBSCRIPT IN COMMON DIMENSIONING.
B020	SUFFIX 5,6 OR 7 NOT ON-TYPE OTHER-NAME.
B021	TYPE OTHER 5,6 OR 7 DOUBLY DEFINED.
B022	ELEMENT LENGTH DESIGNATOR NOT (S) OR (/F).
B023	LEFT,RIGHT PARENTHESIS OR COMMA MISSING IN EQUIVALENCE.
B024	TYPE OTHER 5,6 OR 7 APPEARING WITH SUBSCRIPTS.
B025	THIS EQUIVALENCE CAUSES A REORIGIN OF THE COMMON BLOCK
B026	FORMAL PARAMETER OR ADJUSTABLE DIMENSION IN EQUIVALENCE.
B027	NON-CONSTANT SUBSCRIPT IN EQUIVALENCE.
B030	DECLARED VARIABLE APPEARING IN EXTERNAL STATEMENT.
B031	COMMON/EQUIVALENCE ERROR.
B032	LEFT/RIGHT PARENS NOT MATCHING.
B033	IMPLIED-DO ERROR IN DATA STATEMENT NO = AFTER DO VARIABLE OR, NON-CONSTANT DO LIMITS. OR DO VARIABLE DOES NOT AGREE WITH SUBSCRIPT
B034	NO = AFTER IDENTIFIER.
B035	A VARIABLE APPEARS WITH SUBSCRIPTS BUT HAS NOT BEEN DIMENSIONED
B036	DATA TO ADJUSTABLE DIMENSIONED OR PARTIAL WORD ARRAY.
B037	MULTIPLE DATA TO NON-DIMENSIONED VARIABLE.
B040	DUPLICATE BLOCK NAME.
B041	EQUIVALENCE OVERLAPS COMMON BLOCKS.
B042	FORMAL PARAMETER APPEARS IN COMMON DECLARATION.
B043	VARIABLE NAME GREATER THAN 8 CHARACTERS OR NO COMMA SEPARATOR.
B044	NON-CONSTANT DATA IN LIST.
B046	REPEAT COUNT MUST BE AN INTEGER CONSTANT 1-32767
B050	(S) IS NOT AN INTEGER 1 THRU 7 OR (/F) IS NOT A DIVISOR OF 48
B051	ONE OF THE VARIABLES HAS BEEN DEFINED IN A PREVIOUS TYPE STATEMENT
B052	DOUBLY DEFINED FORMAL PARAMETER
B053	MORE THAN 63 FORMAL PARAMETERS
B201	COMMA MISSING IN PARAMETER LIST OR VARIABLE MORE THAN 8 CHARACTERS.

B202	IMPROPER USE OF FUNCTION NAME.
B203	ILLEGAL SEQUENCE OR USE OF OPERATORS
B204	MIXED MODE-TYPE 5 AND/OR 6 AND/OR 7.
B206	ILLEGAL OPERATOR OR MISSING OPERATOR.
B207	ILLEGAL REPLACEMENT IN ARITHMETIC STATEMENT
H002	AN ARITHMETIC STATEMENT FUNCTION MAY NOT CALL ITSELF
H004	ARITHMETIC STATEMENT FUNCTION DOUBLY DEFINED.
H005	EXTERNAL SYMBOL USED AS ARITHMETIC STATEMENT FUNCTION.
H007	TOO MANY REPLACEMENT OPERATORS IN AN ARITHMETIC STATEMENT FUNCTION.
H010	ILLEGAL PARAMETER LIST FOR ARITHMETIC STATEMENT FUNCTION
H011	ARITHMETIC STATEMENT FUNCTIONS MUST HAVE PARAMETERS.
H013	ILLEGAL PARAMETERS IN ARITHMETIC STATEMENT FUNCTION.
H015	VARIABLE INDEXING IS NOT PERMITTED IN ARITHMETIC STATEMENT FUNCTIONS.
H016	NON-STANDARD INDEXING IS NOT ALLOWED IN ARITHMETIC STATEMENT FUNCTIONS
H017	VARIABLE IDENTIFIER USED AS ARITHMETIC STATEMENT FUNCTION.
H023	THE PARAMETER OF THIS STATEMENT MUST BE TYPE INTEGER.
H024	I IS OUTSIDE THE PERMITED RANGE.
H025	STATEMENT NUMBER IS OUT OF RANGE.
H026	UNIT NUMBER MUST BE A SIMPLE INTEGER VARIABLE OR AN INTEGER CONSTANT.
H030	UNIT NUMBER MUST BE FOLLOWED BY).
H031	AN IF UNIT STATEMENT MUST HAVE 2-4 BRANCH POINTS.
H100	STATEMENT NUMBER IS OUT OF RANGE.
H101	BRANCH POINT ERROR IN IF STATEMENT.
H102	LOGICAL IF IS FORMED INCORRECTLY.
H103	TWO OR MORE RELATIONAL OPERATORS IN THE SAME RELATIONAL SUB-EXPRESSION.
H104	LOGICAL EXPRESSION INCORRECTLY FORMED
H105	RELATIONAL SUB-EXPRESSION FORMED INCORRECTLY.
H106	THE .NOT. OPERATION MUST BE FOLLOWED BY EITHER (OR AN OPERAND.
H112	LOGICAL CONNECTIVE MUST BE FOLLOWED BY (OR AN OPERAND.
H113	A LOGICAL SUBEXPRESSION MAY NOT BEGIN WITH AN OPERATOR
H114	EXCESS LEFT PARENTHESIS IN LOGICAL EXPRESSION.
H200	MASKING ARITHMETIC EXPRESSION TOO LONG.
H201	ARITHMETIC SUB-EXPRESSION IN MASKING STATEMENT NOT FULLY PARENTHESIZED.
H202	FUNCTION CALLED INCORRECTLY.
H210	OPERAND MAY BE FOLLOWED BY OPERATOR OR) ONLY.
H211	.NOT. MUST BE FOLLOWED BY (OR AN OPERAND
H220	THE REPLACEMENT VARIABLE FOR AN EXPRESSION USING LOGICAL OPERATORS MUST BE LOGICAL IF THE STATEMENT IS LOGICAL, OR REAL OR INTEGER IF IT IS MASKING
H212	THE FIRST ELEMENT OF A BOOLEAN EXPRESSION MUST BE AN OPERAND, (OR .NOT.
H213	) MAY BE FOLLOWED ONLY BY .AND., .OR.,).
H214	THE OPERATORS .AND., .OR. MUST BE FOLLOWED BY EITHER (, .NOT., OR AN OPERAND
H215	MASKING OPERANDS MUST BE REAL OR INTEGER
C001	ILLEGAL MARK IN COLUMN SIX.
C002	UN-RECOGNIZED STATEMENT
C011	TOO MANY CHARACTERS IN IDENTIFIER
C016	STATEMENT TOO LONG
C017	UN-MATCHED PARENTHESES
C020	ILLEGAL USE OF BOOLEAN OR RELATIONAL OPERATOR

C025	IMPROPER LENGTH FOR HOLLERITH CONSTANT
C026	ILLEGAL USE OF PERIOD
C027	ILLEGAL CONSTANT TYPE
C030	STATEMENT ENDS WITH ASTERISK
C040	TOO MANY SUBSCRIPT INDICES
C041	ADJACENT COMMAS
C042	RIGHT PAREN PRECEDED BY COMMA
C043	LEFT PAREN FOLLOWED BY COMMA
C044	EMPTY PARENTHETICAL EXPRESSION
C045	LIMIT FOR NON-STANDARD SUBSCRIPT EXPRESSIONS EXCEEDED
C046	NUMBER OF CONSTANTS EXCEEDS COMPILER LIMIT
C047	SUBSCRIPT ON NON-DIMENSIONED VARIABLE
C053	LIMIT FOR STANDARD INDEX FUNCTIONS EXCEEDED
C054	= WITHIN PARENTHESES MAY ONLY APPEAR IN DATA OR I/O LISTS
G001	PARENTHESIS USAGE OR DO LOGIC OR TYPE IDENTIFIER IS ILLEGAL IN I/O DATA LIST.
G002	WRONG FORMAT OF I/O STATEMENT. DATA LIST WAS NOT YET PROCESSED
G003	TAPE NUMBER IN I/O STATEMENT IS GREATER THAN 64
G004	PARITY IN I/O STATEMENT IS NOT BETWEEN 0 AND 2
G005	ILLEGAL SUBSCRIPT IN I/O DATA LIST.
G006	INPUT OF DATA INTO A CONSTANT IS ILLEGAL.
G007	TRANSMISSION OF BYTE SIZED DATA IN BINARY MODE IS ILLEGAL
W003	TYPE OTHER INTERMIXED IN ARITHMETIC.
W004	LOGICAL OR BYTE SIZED OPERAND(S) USED IN EXPONENTIATION.
W005	IMPROPER OPERAND.

INFORMATIVE DIAGNOSTICS -

S011	THE CORRECT FORM FOR THE ENTRY STATEMENT IS ENTRY NAME
S012	ENTRY STATEMENTS SHOULD NOT BE LABELLED
S013	MAIN PROGRAMS SHOULD NOT CONTAIN ENTRY STATEMENTS
S016	THERE IS NO PATH TO THIS STATEMENT
S030	THIS FORMAT STATEMENT IS UNLABELLED
B001	PROGRAM, SUBROUTINE OR FUNCTION CARD NOT FIRST CARD OF DECK.
B045	DOUBLY DEFINED VARIABLE IN COMMON
B210	AN * HAS BEEN INSERTED FOR THE APPEARANCE OF N(,)(,)V OR)N
C003	ASSUMED DIMENSION STATEMENT
C004	ASSUMED BACKSPACE STATEMENT
C005	ASSUMED WRITE-TAPE STATEMENT
C010	ASSUMED WRITE-OUTPUT-TAPE STATEMENT
C007	ASSUMED READ-INPUT-TAPE STATEMENT
C005	ASSUMED WRITE-TAPE STATEMENT
C006	ASSUMED SUBROUTINE STATEMENT
C007	ASSUMED READ-INPUT-TAPE STATEMENT
C010	ASSUMED WRITE-OUTPUT-TAPE STATEMENT

C012	ASSUMED SENSE-LIGHT STATEMENT
C014	ASSUMED IF-OVERFLOW-FAULT STATEMENT
C015	ASSUMED IF-EXPONENT-FAULT STATEMENT
C021	ASSUMED IF-SENSE-LIGHT STATEMENT
C022	ASSUMED IF-SENSE-SWITCH STATEMENT
C023	ASSUMED BUFFER OUT STATEMENT
C024	ASSUMED EQUIVALENCE STATEMENT
C031	ILLEGAL CHARACTER IN LABEL FIELD OR ZERO USED AS STATEMENT LABEL (MAY NOT INHIBIT EXECUTION)
C032	CARD HAS LABEL AND MARK IN COLUMN 6- CONTINUATION ASSUMED
C051	LABELLED BLANK STATEMENT-CONTINUE ASSUMED

INDEX

CONTROL DATA SALES OFFICES

ALAMOGORDO • ALBUQUERQUE • ATLANTA • BOSTON • CAPE CANAVERAL
CHICAGO • CINCINNATI • CLEVELAND • COLORADO SPRINGS • DALLAS • DAYTON
DENVER • DETROIT • DOWNEY, CALIFORNIA • HONOLULU • HOUSTON • HUNTSVILLE
ITHACA • KANSAS CITY, KANSAS • LOS ANGELES • MADISON, WISCONSIN
MINNEAPOLIS • NEWARK • NEW ORLEANS • NEW YORK CITY • OAKLAND • OMAHA
PALO ALTO • PHILADELPHIA • PHOENIX • PITTSBURGH • SACRAMENTO
SALT LAKE CITY • SAN BERNARDINO • SAN DIEGO • SEATTLE • WASHINGTON, D.C.

INTERNATIONAL OFFICES

FRANKFURT, GERMANY • HAMBURG, GERMANY • STUTTGART, GERMANY
GENEVA, SWITZERLAND • ZURICH, SWITZERLAND • CANBERRA, AUSTRALIA
MELBOURNE, AUSTRALIA • SYDNEY, AUSTRALIA • ATHENS, GREECE
LONDON, ENGLAND • OSLO, NORWAY • PARIS, FRANCE • STOCKHOLM, SWEDEN
MEXICO CITY, MEXICO, (REGAL ELECTRONICA DE MEXICO, S.A.)
OTTAWA, CANADA, (COMPUTING DEVICES OF CANADA, LIMITED) • TOKYO, JAPAN,
(C. ITOH ELECTRONIC COMPUTING SERVICE CO., LTD.)

CONTROL DATA
CORPORATION
8100 34th AVENUE SOUTH, MINNEAPOLIS, MINNESOTA 55440

Pub. No. 60052900
Revision A

Litho in U.S.A.