CONTROL DATA
CORPORATION

CONTROL DATA®
CYBER 70/MODEL 76 COMPUTER SYSTEM
7600 COMPUTER SYSTEM

FORTRAN RUN, VERSION 2
REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Manual released.
(11-15-71)	
B XNO	Manual revised to reflect current software. The following pages have been revised: v, vi, vii,
(2-15-73)	2-1, 2-2, 2-3, 2-4, 2-5, 2-7, 2-8, 2-9, 2-11, 3-1, 3-5, 3-6, 3-7, 3-9, 5-1, 5-7, 5-9, 5-15,
	5-16, 6-1, 6-3, 6-4, 6-6, 6-9, 6-14, 7-12, 7-13, 7-14, 7-15, 7-16, 7-18, 8-2, 9-5, 9-6, 9-9,
	9-19, 9-21, 9-22, 10-1, 10-2, 10-3, 10-6, 10-7, 10-8, 10-9, 10-10, 10-11, 10-12, A-1, A-2,
	A-3, A-4, A-5, A-6, A-7, A-8, B-2, B-3, C-2, C-3, E-1, E-2, E-3, E-9, E-10, F-1, G-1,
	G-2, G-5, I-2, K-1, L-2, L-3, L-4, L-6, L-9, L-10, L-11, L-15, L-16, Index 1, 2, 3, and 4.
	Delete pages G-6 and G-7; add pages M-1 and M-2.
C	Manual revised to reflect current software. The following pages have been revised: v, vi, 5-15,
(5-1-74)	9-4 through 9-7, 9-9, 9-11 through 9-18, 9-21, 9-22, 9-23, 9-25, 10-3, 10-7 through 10-10, 10-12,
	E-1 through E-4, L-1, L-3 through L-7, L-9 through L-12, L-14, L-15, L-16, M-1, M-2,
	Index-3, and Index-4.
Publication No. 60360700	

Address comments concerning this manual to:

Control Data Corporation
Technical Publications Department
4201 North Lexington Ave.
Arden Hills, Minnesota 55112

or use Comment Sheet in the back of this manual.

CONTENTS

APPENDIXES

TABLES

INTRODUCTION

FORTRAN for the CONTROL DATA® CYBER 70/Model 76 Computer System is a procedural language designed to solve problems of a mathematical or scientific nature. It incorporates a majority of the features of standardized FORTRAN as specified by the X3.9-1966 American National Standards Institute (ANSI). The variations, as well as the extensions, are flagged throughout this manual.

Design extensions include selected FORTRAN 2.3 features as implemented on the CONTROL DATA® 6000 Series Computers, new source statements relating to the dual memory of the CONTROL DATA® 7600 Computer[†] and features.

LANGUAGE FEATURES

- Constants and variables of the following types:

  Integer
  Single precision floating point (real)
  Double precision floating point
  Complex
  Logical
  Octal (constant only)
  Hollerith (constant only)

- Mixed mode arithmetic expressions
- Masking (Boolean), logical, and relational operators
- Shorthand notation for logical operators and constants
- Library functions (intrinsic functions and external functions)
- Independently compilable subprograms
- Multiple entry points to subroutines and functions
- Expressions as subscripts
- Formatted/unformatted input/output
- Variable dimensions
- Variable format capability

[†] In this publication, any references to the CONTROL DATA CYBER 70/Model 76 Computer System applies also to the 7600 Computer System.

- Intermixed COMPASS subprograms
- Block transfers between large core and small core memory
- Direct reference of lower core memory data
- Allocation of large core memory
- Conversion formats for all data forms
- Array reference with fewer subscripts than dimensioned
- Hollerith constants in expressions and DATA statements
- More than one statement per line
- Left-or right-justified Hollerith constants
- Two-branch logical IF statements
- Two-branch arithmetic IF statements
- NAMELIST capability
- Buffer in/buffer out and encode/decode statements
- Overlay capability
- Multiple assignment statement form
- DATA statement usable in main program
- Abbreviated forms of DATA statement
- DO loop indexing parameters changeable within loop

COMPILER FEATURES

CONTROL DATA 70/Model 76 FORTRAN is a one-pass compiler. It uses both large core memory and small core memory and takes advantage of the 48-parcel instruction stack and segmented functional units. Optimization of DO loops is accomplished by:

- Performing multiple-dimension index calculation before entering the loop
- Evaluating common subexpressions only once
- Evaluating invariant subexpressions before entering the loop

Compiler options available are:

- Compile and execute (no listing)
- Compile with source listing (no execute)
- Compile with source listing and object code output (no execute)
- Compile with source and object listing (no execute)
- Compile with source and object listing and object code output (no execute)
- Compile, produce binary, source, and execute
- File name for compiler input source
- File name for compiler output listing
- File name for compiler object code output
- Line limit on compiler output listing
- Cross-reference listing
- 17/21 bit address mode for LCM data
- Error traceback

The compiler and execution time routines execute under CONTROL DATA CYBER 70/Model 76 SCOPE 2. Subprograms are compiled independently, and a file consisting of relocatable binary subprograms is produced. Upon option, the compiler also produces a source listing, an object code listing, a cross-reference listing, and a relocatable binary deck.

The compiler can execute as a load-and-go program and can produce CONTROL DATA CYBER 70/Model 76 machine language output. It executes as an independent program under control of the operating system and uses only the storage required for compilation of a particular program. Overlays can be loaded at execution time without relocation.

FORTRAN accepts main programs and subprograms written in either FORTRAN source language or COMPASS assembly language. This feature permits a flexible program arrangement for each particular job.

CODING PROCEDURES 1

1.1 CODING LINE

A FORTRAN coding line contains 80 columns in which FORTRAN characters are written one per column. The four types of coding lines are as follows:

Statement	1-5	statement label
	6	blank or zero
	7-72	FORTRAN statement
	73-80	identification field
Continuation	1-5	blank
	6	FORTRAN character other than blank or zero
	7-72	continued FORTRAN statement
	73-80	identification field
Comment	1	C or $ or *
	2-80	comments
Data	1-80	data

ANSI FORTRAN, X3.9-1966, does not specify the identification field or the use of $ and * in comment lines.

1.1.1 STATEMENT

Statement information is written in columns 7 through 72. Statements longer than 66 columns may be continued on the next line. Blanks are ignored by the FORTRAN compiler except in Hollerith fields. The character $ may be used to separate statements when more than one is written on a coding line. However, it cannot be used with a program name, a subroutine name, a function statement, or any statement which requires a statement number. A blank card may be used to separate statements.

Example:

```
I=10$  JL1M=1$  K=K+1$  GO TO 10 is equivalent to:

     I=10
     JL1M=1
     K=K+1
     GO TO 10
```

ANSI FORTRAN, X3.9-1966, does not specify $.

1.1.2 CONTINUATION

The first line of every statement must have a blank or zero in column 6, except statements preceded by $. If statements occupy more than one line, all subsequent lines must have a FORTRAN character other than blank or zero in column 6. Continuation cards may be separated by cards whose first 72 columns are blank. A statement may have up to 19 continuation lines.

1.1.3 STATEMENT LABEL

A statement label is a string of 1 to 5 digits occupying any column position 1 through 5. It serves as a reference to that particular statement. Only statements referred to elsewhere in the program require statement labels. These references can only be executable statements and format statements. The same statement label cannot be given to more than one statement in a program unit. Blanks and leading zeros are ignored. End statements and statements preceding $ cannot have statement labels.

1.1.4 IDENTIFICATION FIELD

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

ANSI FORTRAN, X3.9-1966, does not specify the identification field.

1.1.5 COMMENTS

Each line of comment information is designated by a C, *, or $ in column 1. Comment information appears in the source program and the source program listing, but it is not translated into object code. The continuation character in column 6 is not applicable to comment cards.

ANSI FORTRAN, X3.9-1966, does not specify * or $ or allow comments to appear between continuation lines of a FORTRAN statement.

1.2 PUNCHED CARDS

Each line of the coding form corresponds to one 80-column card; the terms "line" and "card" are often used interchangeably. Source programs and data can be read into the computer from cards; a relocatable binary deck or data can be punched onto cards.

1.3 ORDER OF STATEMENTS

Appendix B, FORTRAN STATEMENT LIST, summarizes the set of skeleton statement forms used in FORTRAN and states whether each individual statement type is executable or not. The entire set of nonexecutable statements (excluding FORMAT statements) present in a program must precede the entire set of executable statements. Comments are not statements and may appear anywhere in the program.

ELEMENTS OF FORTRAN 2

2.1 FORTRAN CHARACTER SET

Forty-seven characters are used in forming the elements of a FORTRAN program. These are divided into alphanumeric characters and special characters. The alphanumeric characters are the alphabetic capital letters from A to Z and the decimal numerics from 0 to 9. The special characters are the following:

	blank	(	left parenthesis
=	equal	)	right parenthesis
+	plus	,	comma
-	minus	.	decimal point
*	asterisk	$	dollar sign
/	slash		

Appendix A includes a list of additional characters which may appear in Hollerith literals and with the exception of the semicolon, in DATA statements.

All characters appear internally in display code (Appendix A). A blank is ignored by the compiler except in Hollerith fields. It may be used freely to improve program readability.

2.2 SYMBOLIC NAMES

Symbolic names are used to identify data, programs, subprograms, I/O units, and labeled common blocks. With one exception, a symbolic name can be any combination of one to seven alphanumeric characters beginning with a letter.

> ANSI FORTRAN, X3.9-1966, limits all symbolic names to six characters.

The exception is a form of the octal constant which is the letter O followed by six octal digits. Embedded blanks in a symbolic name are ignored.

> ANSI FORTRAN, X3.9-1966, allows symbolic names to consist of the letter O followed by a string of digits.

Example:

Illegal Symbolic Names		Legal Symbolic Names
3BETA	Begins with numeric character	IOTA
REMAINDER	More than seven characters	A123456
+ 234	Begins with special character	O12345
O123456	Illegal as a symbolic name but legal as an octal constant	O12K345
C3114	Contains a special character	

2.3 DATA TYPES

The seven different data types specified for CONTROL DATA CYBER 70/Model 76 FORTRAN are integer, real, double precision, complex, logical, Hollerith, and octal. Implicit declaration of type is applicable to integer and real only. In the case of a constant, the absence of a decimal point indicates type integer, and the presence of a decimal point indicates type real. In the case of a variable, an I, J, K, L, M, or N as an initial letter indicates type integer. Any other alphabetic character used as an initial letter indicates type real. Double precision, complex, and logical data must be declared in a type statement. Hollerith and octal constants are treated as type integer when they appear in arithmetic expressions or assignment statements.

2.4 CONSTANTS

A constant can be any of the seven data types listed above. Complex and double precision constants are formed from real constants. The type of constant is determined by its form. The computer word structure for each type is given in Appendix D.

2.4.1 INTEGER CONSTANTS

An integer constant is a string of up to 18 decimal digits in the range $-(2^{59}-1) \leq N \leq (2^{59}-1)$. The maximum value of the result of integer addition or subtraction must not exceed $2^{59}-1$. Subscripts and DO indexes are limited to 2^{17-2}. The constant must not contain embedded commas.

Examples:

63	3647631	-314159265	574396517802457165
247	464646464		

During execution, the maximum allowable value is $2^{48}-1$ when an integer constant is converted to real. If the result is greater than $2^{48}-1$, bits 48-58 will be ignored and errors may result. The maximum value of the operands and the result of integer multiplication or division must be less than $2^{48}-1$. High order bits will be lost if the value is larger, but no diagnostic is provided for values greater than $2^{48}-1$. The high order bits are lost.

2.4.2 REAL CONSTANTS

A real constant is a signed or unsigned string of up to 14 decimal digits that includes a decimal point and/or an exponent. All real numbers are carried in normalized form.

A real constant has one of the following forms:

n.m	n.mE±s	nE±s	nEs
n.	n.E±s	n.Es	n.mEs
.m	.mE±s	.mEs	

Where n and m are decimal, s is the exponent to the base 10, and E is the symbol used to indicate exponentiation. The plus sign may be omitted for positive s. The range of a non-zero constant is approximately 10^{-294} to 10^{+322}. If the range is exceeded, a compiler diagnostic is provided. If the magnitude is less than 10^{-294}, the value will be zero.

Examples:

3.E1 (means 3.0 x 10^{1};i.e., 30.)

3.1415768   31.41592E-01

314.0749162   .31415E01

-3.141592E+279   .31415E+01

2.4.3 DOUBLE PRECISION CONSTANTS

A double precision constant is a signed or unsigned string of up to 29 decimal digits that includes a decimal point. It is optionally followed by an exponent. It is represented internally by two words. The forms are similar to real constants:

.mD±s   n.mD±s   n.D±s   nD±s

.mDs   n.mDs   n.Ds   nDs   .mD   n.mD   n.D   nD

Where n and m are decimal, s is the exponent to the base 10, and D is the symbol indicating double precision. D must always appear. The plus sign may be omitted for positive s. The range of non-zero constant is, approximately, from 10^{-294} to 10^{+322} (double precision values between 10^{-294} and 10^{-279} have only single precision with the least significant word set to zero). If the range is exceeded, a compiler diagnostic is provided. If s is omitted, is is assumed to be zero.

Examples:

3.1415927D   3141.593D3

3.1416D0   31416.D-04

3141.593D-03

2.4.4 COMPLEX CONSTANTS

A complex constant appears as an ordered pair of optionally signed real constants.

The form is:

(r_1, r_2)

where the real part of the complex number is represented by r_1 and the imaginary part by r_2.

ANSI FORTRAN, X3.9-1966, does not allow the omission of s.

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

Examples:

FORTRAN 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.)	0 - 1.0i

2.4.5 LOGICAL CONSTANTS

A logical constant is a truth value. It may assume only the value of true or the value of false. A true constant is stored internally as the one's complement of binary zero. A false constant is stored internally as binary zero.

The two permissible forms of a logical constant are:

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

or the briefer alternate forms

```
.T.
.F.
```

The latter forms are not specified in ANSI X3.9 FORTRAN.

Examples:

```
LOGICAL X1, X2
X1 = .TRUE.
X2 = .FALSE.
```

2.4.6 HOLLERITH CONSTANTS

A Hollerith constant is a string of FORTRAN characters which is represented in memory by an internal display code and is treated as an integer.

The general form is:

$nHh_1h_2\ldots h_n$

where n is an unsigned decimal integer indicating the number of characters following H which are part of the constant. H is the symbol indicating Hollerith type. The h_i are the FORTRAN characters that make up the constant. h_i may be any of the characters listed in Appendix A, with the exception of the semicolon.

Blanks are significant.

The maximum number of characters allowed in a Hollerith constant of H form depends on its usage. When used in an expression, it is limited to 10 characters. In a DATA statement, or when passed as an actual argument to a subprogram, it is limited only by the necessity that the statement containing it be limited to 19 continuation lines. The long Hollerith constant must be dimensioned in a subprogram when used as an argument.

ANSI FORTRAN, X3.9-1966, allows Hollerith constants only in the argument list of a CALL statement and in the DATA statement.

Alternate forms of the Hollerith constant are:

$nLh_1, h_2 \ldots h_n$ (left justified)

$nRh_1, h_2 \ldots h_n$ (right justified)

Both left and right justification are with binary zero fill. If more than 10 characters are used in a DATA statement involving such a constant, only the last word has the zero fill. These forms may be used in an arithmetic statement such as in the statement I =(+5HABCDE).

ANSI FORTRAN, X3.9-1966, does not specify the alternate forms nLh and nRh, $nLh_1h_2 \ldots h_n$ and $nRh_1h_2 \ldots h_n$.

Examples:

6HCOGITO

4HERGO

3HSUM

5RSUMbb=00000SUMbb

12HCONTROL DATA

5LSUMbb=SUMbb00000

1H)

3LbTT=bTT0000000

A semicolon (display code 77) cannot appear in Hollerith constants since this bit configuration is recognized as a Hollerith field terminator. When a Hollerith constant is stored, neither the nH, nL, nor nR character is shared with it.

2.4.7 OCTAL (MASKING) CONSTANTS

An octal constant is an optionally signed string of octal digits. It may have a minus sign prefix. It is considered type integer.

The two forms of the octal constant are:

$On_1 \ldots n_i$ $6 \leq i \leq 20$

$n_1 \ldots n_iB$ $1 \leq i \leq 20$

The first form consists of 6 to 20 octal digits preceded by the letter O. The second form consists of 1 to 20 octal digits followed by the letter B.

Octal constants are right justified with zero fill. If the constant exceeds 20 digits, or if a non-octal digit appears, a compiler diagnostic is provided.

ANSI FORTRAN, X3.9-1966, does not specify octal constants.

Examples:

```
O77777770007777          2374216B
O777777700077777         777776B
O2323232323232323        777000777000777B
```

2.5 VARIABLES

FORTRAN recognizes simple and subscripted variables. A simple variable represents a single quantity and references a storage location. The value specified by the name is always the current value stored in the location. Variables are identified by a symbolic name as defined in paragraph 2.2.

The compiler does not check to see whether a variable has been assigned a value. The user must make certain that all variables are defined. Otherwise, unexpected values may result.

The type of a variable is defined in one of two ways:

Explicit — Variables may be declared a particular type with the FORTRAN type declarations (paragraph 5.1).

Implicit — A variable not defined in a FORTRAN type declaration is assumed to be integer if the first character of the symbolic name is I, J, K, L, M, or N.

Example:

I15, JK26, KKK, NP362L, M

All other variables not declared in a FORTRAN type declaration are assumed to be real.

Example:

TEMP, ROBIN, A55, R3P281

2.5.1 INTEGER VARIABLES

Integer variables are defined explicitly or implicitly. They may assume values in the range

$$-(2^{59}-1) \leq I \leq (2^{59}-1).$$

The maximum absolute value a particular integer variable may assume depends on usage. The result of conversion from integer to real, of the integer multiplication, integer division, or input/output under the I-format specification is limited to 2^{48}-1. The result of integer addition or subtraction can be as great as 2^{59}-1. Subscripts and DO indexes are limited to 2^{17}-2. The range of values and number of significant digits are the same as for integer constants described in paragraph 2.4.1. Each integer variable occupies one word of storage.

Examples:

IOTA
J
K2S04
LLLLLL
M58A
NEGATE

2.5.2 REAL VARIABLES

Real variables are defined explicitly or implicitly. They may assume values in the range

$$10^{-294} < |X| < 10^{+322}$$

with approximately 14 significant digits.

More specifically, X may assume the following values:

$$-10^{+322} < X < -10^{-294}$$

$$X = 0$$

$$10^{-294} < X < 10^{+322}$$

The range of values and number of significant digits are the same as for real constants described in paragraph 2.4.2. Each real variable is stored in floating-point format and occupies one word.

Examples:

ALPHA
BETA
GAMMA
XXXX
Z62597
REAL22

2.5.3 DOUBLE PRECISION VARIABLES

Double precision variables must be defined explicitly by a type declaration. The range of values and number of significant digits are the same as for double precision constants described in paragraph 2.4.3. Each double precision variable occupies two words of storage and can assume values in the range $10^{-294} \leq |d| \leq 10^{+322}$ with approximately 29 significant digits. Essentially, the double precision variable is a real variable with storage extended in order to achieve greater precision.

Example:

```
DOUBLE PRECISION OMEGA, X, IOTA
```

A double precision real variable is generated for each of the three symbolic names, OMEGA, X, and IOTA.

2.5.4 COMPLEX VARIABLES

Complex variables must be explicitly defined by a type declaration. A complex variable occupies two words in storage. Each word contains a number in real variable format. This ordered pair of real variables (C_1, C_2) represents the complex number(C_1+C_2i). Note that a complex variable in FORTRAN cannot actually be written as (C_1, C_2) since C_1 and C_2 must be constants when this form is used. The correct form is CMPLX (C_1, C_2) when either C_1 or C_2 is a variable. (See Appendix C for information about CMPLX and paragraph 2.4.4 for information about complex constants.)

Example:

```
COMPLEX ZETA, MU, LAMBDA
```

A pair of real variables comprising a complex variable is generated for each of the three symbolic names, ZETA, MU, and LAMBDA.

2.5.5 LOGICAL VARIABLES

Logical variables must be defined explicitly by a type declaration. Each logical variable occupies one word of storage. It can assume the value of true or false. A logical variable with a positive zero value is false. Any other value is true. When a logical variable appears in an expression whose dominant mode is real, double, or complex, it is not packed and normalized prior to its use in the evaluation of an expression (as is the case with an integer variable).

Example:

```
LOGICAL VALUE, L33, PRAVDA
```

A logical variable is generated for each of the three symbolic names, VALUE, L33, and PRAVDA.

2.6 SUBSCRIPTED VARIABLE

A subscripted variable may have one, two, or three subscripts enclosed in parentheses. More than three produce a compiler diagnostic. Subscripts can be expressions in which the operands are simple integer variables and integer constants. The operators are addition, subtraction, multiplication, and division only. Such expressions must result in positive integers. Use of other values such as zero, real, negative integer, complex, or logical may invalidate results.

When a subscripted variable represents the entire array, the subscripts are the dimensions of the array. When a subscripted variable references a single element in an array, the subscripts describe the relative location of the element in the array.

> ANSI FORTRAN, X3.9-1966, allows only the forms c*v ± k, c*v, v ± k, v, k for subscript expressions, where c and k are unsigned integer constants and v is an integer variable.

Valid Subscripted Variables	Invalid Subscripted Variables
A(I, J)	FRAN (0)
B(I+2, J+3, 2*K+L)	P(3. 5)
Q(14)	Z14(-4)
STRING (3*K*ILIM+3)	EVAL(2+(3. 1, 2. 5)
Q(1, 4, 2)	I(2, -5, 3)

2.7 ARRAYS

An entire array, a block of successive storage locations, may be referenced by the array name without subscripts (in I/O list, data statements, call statements, and function references). Arrays may have one, two, or three dimensions. The array name and dimensions must be declared in a DIMENSION, COMMON, or TYPE declaration prior to the first program reference to that array.

Each element in an array may be referenced by the array name plus a subscript notation. Program execution errors may result if subscripts are larger than the dimensions initially declared for the array. The maximum number of elements in an array is the product of the dimension.

Array elements are stored by columns in ascending locations.

In the array declared as A(3,3,3):

A_{111} A_{121} A_{131}

A_{211} A_{221} A_{231}

A_{311} A_{321} A_{331}

A_{112} A_{122} A_{132}

A_{212} A_{222} A_{232}

A_{312} A_{322} A_{332}

A_{113} A_{123} A_{133}

A_{213} A_{223} A_{233}

A_{313} A_{323} A_{333}

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

A_{111} → L A_{121} → L+3 .... A_{133} → L+24
A_{211} → L+i A_{221} → L+4 .... A_{233} → L+25
A_{311} → L+2 A_{321} → L+5 .... A_{333} → L+26

Array allocation is discussed under DIMENSION declaration. 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 enclosed by the outer parenthesis is the subscript expression. E is the element length, the number of storage words required for each element of the array. For real, logical, and integer arrays, E=1. For complex and double precision arrays, E=2.

Example:

In an array defined by DIMENSION A(3,3,3) the location of A(2,2,3) with respect to A(1,1,1) is:

Locn A(2,2,3) = (Locn A(1,1,1) + (2-1+3(2-1+3(3-1)))*1
= (L + 22)*1 = L + 22

An array reference is not checked by the compiler to see whether it is within the limits of the array as defined.

CONTROL DATA CYBER 70/Model 76 FORTRAN permits the following relaxation of 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, 1)
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(I) implies A(I)
A implies A(1)[†]

The elements of a single-dimension array $A(D_1)$ may not be referred to as A(I, J, K) or A(I, J). Similarly, the elements of the double-dimension array $A(D_1, D_2)$ may not be referred to as A(I, J, K). Diagnostics occur if this is attempted.

ANSI FORTRAN, X3.9-1966, does not specify the above relaxations.

[†] In I/O lists or in DATA statements, A by itself implies the whole array.

EXPRESSIONS 3

An expression is a set of operands combined by operators and parentheses to produce, at time of execution, a single-value result. The set may be a single character or it can be a complex string of operands and operators nested within parentheses. There are four kinds of expressions in FORTRAN: arithmetic, masking (Boolean), logical, and relational. Arithmetic and masking expressions produce numerical results. Logical and relational expressions produce truth values. Each type of expression is associated with particular sets of operators and operands.

3.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of arithmetic operators and operands which, when evaluated, produces a single numerical value.

The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

The arithmetic operands are:

Constants

Variables (simple or subscripted)

Function References

Examples:

```
A
3.14159
3 + 16.427
(XBAR + (B(I,J+1,K)/3))
-(C + DELTA * AERO)
(B - SQRT (B**2 - (4*AC)))/(2.0*A)
```

3.1.1 FORMING ARITHMETIC EXPRESSIONS

Two arithmetic operators cannot be adjacent in an arithmetic expression. If minus is used to indicate a negative operand, the sign and the element must be enclosed in parentheses if preceded by an operator.

Parentheses may be used to indicate grouping as in ordinary mathematical notation. They may not be used to indicate multiplication. Parentheses must always be used in pairs.

Illegal	Legal
B*-A	A*B-C
X/-Y*Z	B*A/(-C)
R(-S)	A*(-C)

Any arithmetic operand or expression may be raised to a power that is a positive or negative integer operand or expression.

Examples:

```
M**N
(X+Y)**I
(A+B)**(-J)
IVAL**(K+2)
```

Only a positive real operand or expression can be raised to a real power.

```
ALPHA**3.2
(X+Y)**A
(A+B)**(+3.)
```

When writing an integer expression, it is important to remember that to divide an integer quantity by an integer quantity will always yield a truncated result.

Example:

3/2 * 4 → 4 rather than 6

3.1.2 ARITHMETIC EVALUATION

Parenthetical and functional expressions are evaluated first in a right-to-left scan. In parenthetical expressions within parenthetical expressions, evaluation begins with the innermost expression and proceeds outward. Separate parenthetical expressions are evaluated as they are encountered in the right-to-left scan.

In an expression with no parentheses or an expression within a pair of parentheses in which unlike operators appear, evaluation proceeds according to the following hierarchy of operators:

**	exponentiation	first
/	division	next
*	multiplication	
+	addition	last
-	subtraction	

In an expression with like operators, evaluation proceeds from left to right, that is,

$A/B*C = (A/B)*C$

Examples:

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

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

$A**B \rightarrow R_1$

$R_1/C \rightarrow R_2$

$D*E \rightarrow R_3$

$R_3F \rightarrow R_4$

$R_4+R_2 \rightarrow R_5$

$R_5-G \rightarrow R_6$ Evaluation completed

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

$E*F-G \rightarrow R_1$

$C+D \rightarrow R_2$

$A**B \rightarrow R_3$

$R_3/R_2 \rightarrow R_4$

$R_4*R_1 \rightarrow R_5$ Evaluation completed

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

$COS(Z) \rightarrow R_1$

$R_1**2 \rightarrow R_2$

$R_2*C(I, J+2) \rightarrow R_3$

$R_3+H(13) \rightarrow R_4$ Evaluation completed

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

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

$C/D \rightarrow R_1$

$R_1-E \rightarrow R_2$

$R_2+B \rightarrow R_3$

$R_3*A \rightarrow R_4$ Evaluation completed

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

$E+F \rightarrow R_1$

$D-R_1 \rightarrow R_2$

$C*R_2 \rightarrow R_3$

$SIN(X) \rightarrow R_4$

$R_4+1. \rightarrow R_5$

$A*R_5 \rightarrow R_6$

$R_6-Z \rightarrow R_7$

$R_7/R_3 \rightarrow R_8$ Evaluation completed

3.1.3 MIXED-MODE ARITHMETIC EXPRESSIONS

Mixed-mode arithmetic is permissible for all combinations of types (integer, real, double-precision, complex, and logical operands) using any of the mathematical operations except exponentiation. The type of an evaluated mixed-mode arithmetic expression is the mode of the dominant operand type. The order of dominance of operand types within an expression is given by the following list which proceeds from highest to lowest:

Complex

Double precision

Real

Integer

Logical

ANSI FORTRAN, X3.9-1966, does not specify logical operands in arithmetic expressions. For operators other than exponentiation, the operands may be of the same type as one may be real and the other double precision or complex.

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

If B is preceded by a minus operator, the form is A**(-B).

A and B are treated as integers if type is logical.

For the various operand types, the type relationships of A**B are:

Type B

	**	I	R	D	C	L
Type A	I	I	no †	no	no	I
	R	R	R	D	no	R
	D	D	D	D	no	D
	C	C	no	no	no	C
	L	I	no	no	no	I

† no indicates an invalid operation

For example, if A is real and B is integer, the mode of A**B is real. However, B**A is illegal.

ANSI FORTRAN, X3.9-1966, does not specify exponentiation if either A or B is logical.

Examples:

1. Given real A, B; integer I, J. The type of expression A*B- 1+J is real because the dominant operand type is real.

   The expression is evaluated:

   Convert I to real

   Convert J to real

   $A*B \rightarrow R_1$ real

   $R_1-I \rightarrow R_2$ real

   $R_2+J \rightarrow R_3$ real

2. The use of parentheses can change the evaluation. A,B,I,J are defined as above.

   A*B-(I-J) is evaluated:

   $I-J \rightarrow R_1$ integer

   $A*B \rightarrow R_2$ real

   Convert R_1 to real

   $R_2-R_1 \rightarrow R_3$ real

3. Given complex C,D, real A,B. Thetype of the expression A* (C/D)+B is complex because the dominant operand type is complex.

   The expression is evaluated:

   $C/D \rightarrow R_1$ complex

   Convert A to complex

   $A*R_1 \rightarrow R_2$ complex

   Convert B to complex

   $R_2+B \rightarrow R_3$ complex

4. Consider the expression C/D+(A-B) where the operands are defined in 3 above.

   The expression is evaluated:

   $A-B \rightarrow R_1$ real

   $C/D \rightarrow R_2$ complex

   Convert R_1 to complex

   $R_1+R_2 \rightarrow R_3$ complex

5. Mixed-mode arithmetic with all types is illustrated by this example:

   Given: the expression C*D+R/I-L

C	Complex
D	Double
R	Real
I	Integer
L	Logical

The dominant operand type in this expression is complex; therefore, the evaluated expression is complex.

Evaluation:

Convert D to complex

(Truncate D to real and affix zero imaginary part.)

$C*D \rightarrow R_1$ complex

Convert R to complex

Convert I to complex

$R/I \rightarrow R_2$ complex

$R_2+R_1 \rightarrow R_3$ complex

$R_3-L \rightarrow R_4$ complex

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

Convert I to real

$R/I \rightarrow R_1$ real

$R_1-L \rightarrow R_2$ real

Convert D to complex

$C*D \rightarrow R_3$ real

Convert R_2 to complex

$R_2+R_3 \rightarrow R_4$ complex

3.2 RELATIONAL EXPRESSIONS

A relational expression is a combination of two arithmetic expressions with a relational operator. The relational expression will have the true or false value depending on whether the stated relation is valid or not. A true relational expression is assigned the value minus zero (all one bits). A false relational expression is assigned the value plus zero (all zero bits).

The general form of a relational expression is:

a_1 op a_2

where the a's are arithmetic expressions and op is one of the relational operators.

ANSI FORTRAN, X3.9-1966, specifies only relational expressions for which an arithmetic expression is of type real or double precision and the other is of type real or double precision or both arithmetic expressions are of type integer.

NOTE

A relational expression can have only two operands combined by one operator. The form a_1 op a_2 op a_3 is not valid.

The relational operators are:

Symbol	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

Relational expressions of the following forms where I is integer, R is real, D is double precision and C is complex, are allowed:

I.LT.R	(Mixed mode. Convert I to real)
I.LT.D	(Mixed mode. Convert I to double precision)
I.LT.C	(Mixed mode. Real part of C is used)

A relation of the form a_1 op a_2 is evaluated from left to right.

The relations a_1 op a_2, a_1 op (a_2), (a_1) op a_2, (a_1) op (a_2) are equivalent.

Usually programs will compute and execute faster if the minimum number of parentheses is used.

Examples:

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

Mixed mode is permissible in relational expressions for all combinations of types (integer, real, double precision, and complex). The order of dominance of the operand types is the same as that stated for mixed-mode arithmetic expressions (paragraph 3.1.3). When complex expressions are tested for zero or minus zero, only the real part is used in the comparison. For double precision numbers, the value is converted to real.

> ANSI FORTRAN, X3.9-1966, specifies that the length of the real shall be converted to double precision length for use in evaluating the relational expression.

Relational expressions are converted to equivalent arithmetic expressions at compile time. At execution time, these equivalent arithmetic expressions are evaluated with program-supplied values and compared with zero to determine the truth value of the corresponding relational expression. For example, the relation p.EQ.q is equivalent to p-q=0. The difference is computed and tested for zero at the time of execution. If the difference is zero (or minus zero), the relation is true; otherwise, it is false. Likewise, the relation p.GE.q is equivalent to $p-q \geq 0$. At time of execution, the difference p-q is computed and compared with zero. If the difference is greater than or equal to zero, the relation is true. If the relation is less than zero, it is false.

The relational expression I.GE.0 is treated as true if I assumes the value minus zero or plus zero.

3.3 LOGICAL EXPRESSIONS

A logical expression is a combination of logical operands and/or relational expressions with logical operators which, when evaluated, will have a value of true or false. The general form of a logical expression is:

L_1 op L_2 op L_3. . .

where the L's are logical operands or relational expressions and the op's are logical operators.

The logical operands are:

Logical constant	Either the value .TRUE. or the value .FALSE.
Logical variable	A variable that has been declared in a LOGICAL type statement. It can only assume the values .TRUE. or .FALSE.

The logical operators are:

.NOT.	Logical negation	Reverses the truth value of the logical expression that follows it
.AND.	Logical conjunction	Combines two logical expressions to produce a value of .TRUE. whenever both expressions are true; otherwise, it gives a value of .FALSE.
.OR.	Logical disjunction	Combines two logical expressions to produce a value of .TRUE. whenever either or both expressions are true; otherwise, it gives a value of .FALSE.

Alternate forms of the logical operators are:

.N.

.A.

.O.

ANSI FORTRAN, X3.9-1966, does not specify the alternate forms of the logical operators.

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

.NOT. L_1

The value of the expression is examined. If the value is equal to plus zero, the logical expression has the value false. All other values are considered true.

The hierarchy of logical operations is:

First	.NOT.	or	.N.
then	.AND.	or	.A.
then	.OR.	or	.O.

A logical variable, logical constant, or a relational expression is, in itself, a logical expression. If E_1, L_2 are logical expressions, then the following are logical expressions:

.NOT.L_1
L_1.AND.L_2
L_1.OR.L_2

If L is a logical expression, then (L) is a logical expression.

If L_1, L_2 are logical expressions and op is .AND. or .OR. then L_1 op op L_2 is never legitimate.

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

L_1.AND. .NOT.L_2
L_1.OR. .NOT.L_2
L_1.AND. (.NOT.$\cdots$)
L_1.OR. (.NOT.$\cdots$)

.NOT. may appear with itself only in the form .NOT.(.NOT.(.NOT. L))

Other combinations cause compilation diagnostics.

If L_1, L_2 are logical expressions, the logical operators are defined as follows:

.NOT.L_1	is false only if L_1 is true
L_1.AND.L_2	is true only if L_1, L_2 are both true
L_1.OR.L_2	is false only if L_1, L_2 are both false

Examples:

$B-C \leq A \leq B + C$ is written

B-C.LE.A.AND.A.LE.B+C

FICA greater than 374.40 and PAYNMB equal to 5889.0 is written

FICA.GT.374.40.AND.PAYNMB.EQ.5889.0

An expression equivalent to the logical relationships (P → Q) may be written in two forms:

.NOT.(P.AND.(.NOT.Q))

.N.(P.A.(.N.Q))

3.4 MASKING EXPRESSIONS

The masking expression is a generalized form of the logical expression in which the variables may be types other than logical.

In a FORTRAN masking expression, 60-bit logical arithmetic is performed bit-by-bit on the operands within the expression. The operands may be any type variable, constant, or expression. No mode conversion is performed during evaluation. If the operand is complex, operations are performed on the real part. Although the masking operators are identical in appearance to the logical operators, their meanings are different. They are listed according to hierarchy.

Examples:

.NOT. or .N.	Complement the operand
.AND. or .A.	Form the bit-by-bit logical product of two operands
.OR. or .O.	Form the bit-by-bit logical sum of two operands

ANSI FORTRAN, X3.9-1966, does not specify masking expressions.

The operations are described as follows:

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

Let B_i be masking expressions, variables or constants of any type except logical. The following are masking expressions:

.NOT.B_1 B_1.AND.B_2 B_1.OR.B_2

If B is a masking expression, then (B) is a masking expression.

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

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

Masking expressions of the following forms are evaluated from left to right: ʻ

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

Arithmetic expressions appearing in masking statements must be enclosed in parentheses; for example, E=(E*100B).OR.F.

Examples:

Given:

A	77770000000000000000	octal constant
D	00000000777777777777	octal constant
B	00000000000000001763	octal form of integer constant
C	20045000000000000000	octal form of real constant

Then:

.NOT.A	is	00007777777777777777
A .AND. C	is	20040000000000000000
A .AND..NOT.C	is	57730000000000000000
B .OR..NOT.D	is	77777777000000001763

The last expression could also be written as B .O. .N. D

ASSIGNMENT STATEMENTS 4

4.1 ARITHMETIC ASSIGNMENTS

The general form of the arithmetic assignment 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.

ANSI FORTRAN, X3.9-1966, specifies that if either A or E is complex logical, then both A and E must be complex logical.

Examples:

```
      A=-A
      B(J,4) = CALC(I+1)*BETA+2.3478
39    XTHETA=7.4*DELTA+(A(I,J,K)**BETA)
      RESPSNE=SIN(ABAR(INV+2,JBAR)/ALPHA(J,KAPL(I)))
4     JMAX=19
      AREA = SIDE1*SIDE2
      PERIM = 2.*(SIDE1 + SIDE2)
```

4.2 MIXED-MODE ASSIGNMENT

The type 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. A complex expression may replace A, even if A is real. The following chart shows the A = E relationship for all the standard modes. The mode of A determines the mode of the statement.

	Type of Expression E			
Type of A	Complex	Double Precision	Real	Integer
Complex	$A = E$	Set A = more significant half of E $A_{real} = E$ $A_{imag} = 0$	$A_{real} = E$ $A_{imag} = 0$	Convert E to real $A_{real} = E$ $A_{imag} = 0$
Double Precision	$A = E_{real}$ Less significant is set to zero	$A = E$	$A = E$ Less significant is set to zero	Convert E to real $A = E$ Less significant is set to zero
Real	$A = E_{real}$	Set A = more significant half of E $A = E$	$A = E$	Convert E to real $A = E$
Integer	Truncate E_{real} to integer $A = E$	Truncate E to 48 bit integer $A = E$	Truncate E to integer $A = E$	$A = E$
Logical	If $E_{real} \neq 0$, $A = E_{real}$ If $E_{real} = 0$, $A = 0$	If $E \neq 0$, A=more significant half of E If $E = 0$, $A = 0$	If $E \neq 0$, $A = E$ If $E = 0$, $A = 0$	If $E \neq 0$, $A = E$ If $E = 0$, $A = 0$

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

If

L_1, L_2, L_3	logical variables
R	real variable
I	integer variable

then

$I=L_1*L_2+L_3-L_4$

is evaluated as if the L_i were all integers. The resulting value is 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.

Because of computer hardware, when a mode conversion is made from real, double precision, or complex to integer, and the real number (or real portion of complex number) is in the range $-1< R< 0$, this conversion is made.

Real Value	Resulting Integer
$-1< R< (-(2^{-17}))$	-0
$-(2^{-17})< R< 0$	0

Example:

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

$A_1 = C_1*C_2-C_3/C_4$ (6.905, 15.393) = (4.4, 2.1)* (3.0, 2.0) - (3.3, 6.8) / (1.1, 3.4)

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

$A_3 = C_1$ 4.4000+000 = (4.4,2.1)

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

$A_3 = C_1*(0.,-1.)$ 2.1000+000 = (4.4,2.1)*(0.,-1.)

The expression is complex. A_3 is real; the real part of the result of the complex multiplication replaces A_3.

$A_4 = R_1/R_2*(R_3-R_4)+I_1-(I_2*R_5)$ 13=8.4/4.2*(3.1-2.1) + 14-(1*2.3)

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

$A_2 = D_1**2*(D_2+(D_3*D_4))$ 4.96800000000000+001 = 2.0D**2*(3.2D+(4.1D*1.0D))

$+(D_2*D_1*D_2)$ +(3.2D*2.0D*3.2D)

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

$A_5 = C_1*R_1-R_2+I_1$ 1=(4.4,2.1)*8.4-4.2+14

The expression is complex. Since A_5 is logical, the real part of the evaluated expression replaces A_5. If the real part is zero, zero replaces A_5.

4.3 LOGICAL ASSIGNMENT

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

Examples:

```
LOGICAL A, B, C, D, E, LGA, LGB, LGC
REAL F, G, H
A = B .AND. C .AND. D
A = F .GT. G .OR. F .GT. H
A = .N. (A.A. .N. B) .AND. (C.O.D)
LGA = .NOT. LGB
LGC = E .OR. LGC .OR. LGB .OR. LGA .OR. (A .AND. B)
```

4.4 MASKING ASSIGNMENT

The general form of the masking assignment statement is M = E. E is a masking expression, and M is a variable of any type except logical. No mode conversion is made during thc replacement.

Examples:

```
INTEGER I,J,K,L,M,N(16)
REAL B,C,D,E,F(15)
N(2) = I .AND. 77B
B = C .AND. L
F(J) = I .OR. .NOT. L .AND. F(J)
N(1) = I.O.J.O.K.O.L.O.M
I = .N.I
D = (B.LE.C) .AND. (C.LE.E) .AND. .NOT.I
```

ANSI FORTRAN, X3.9-1966, does not specify masking assignment.

4.5 MULTIPLE ASSIGNMENT

Expressions of the form:

```
A=B=C=D=3.0*X
```

arepermissible and are executed right to left. The above expression would result in a code which is equivalent to the expressions:

```
D=3.0*X
C=D
B=C
A=B
```

ANSI FORTRAN, X3.9-1966, does not specify multiple assignment.

TYPE DECLARATIONS AND STORAGE ALLOCATION 5

Declarative statements are nonexecutable statements which define the arithmetic and allocation attributes and initial values of data. The statements may appear in any order but they must precede any executable statement.

The six types of declarative statements are:

Type
DIMENSION
COMMON
EQUIVALENCE
DATA
LEVEL

5.1 TYPE DECLARATION

The type declaration statement provides the compiler with information on the structure of variable and function identifiers.

Statement	Characteristics	
COMPLEX list	2 words/element	Floating point
DOUBLE PRECISION list or DOUBLE list	2 words/element	Floating point
REAL list	1 word/element	Floating point
INTEGER list	1 word/element	Integer
LOGICAL list	1 word/element	Logical

Type may precede any of the above statements.

ANSI FORTRAN, X3.9-1966, does not specify TYPE as prefix to type declaration statements.

DOUBLE may replace DOUBLE PRECISION in any FORTRAN statement in which the latter is allowed.

> ANSI FORTRAN, X3.9-1966, does not specify DOUBLE as a replacement for DOUBLE PRECISION.

List is a string of identifiers separated by commas: integer constant subscripts are permitted.

Example:

```
    A, B1, CAT, D36F, GAR (1,2,3)
```

The type declaration is nonexecutable and must precede the first executable statement. If an identifier is declared in two or more type declarations, the last declaration holds. Any declaration following the first results in an informative diagnostic if the previously declared type was not the implicit type of identifier.

> ANSI FORTRAN, X3.9-1966, does not specify redeclaration of type.

An identifier not declared in a type declaration is type integer if the first letter of the name is I, J, K, L, M, N; for any other letter it is type real. When subscripts appear in the list, the associated identifier is the name of an array. The product of the subscripts determines the amount of storage to be reserved for that array. Thus, dimension and type information are given in the same statement. In this case, no DIMENSION statement is needed. If a second declaration of storage appears, an informative diagnostic is issued and the original declaration is used.

Standard library functions whose type is not integer or real are known to that compiler and thus they need not appear in a type declaration in the user's program (see Appendix C).

Examples:

```
    COMPLEX A412,DATA,DRIVE,IMPORT
    DOUBLE PRECISION PLATE,ALPHA(20,20),B2MAX,F60,JUNE
    REAL I,J(20,50,2),LOGIC,MPH
    INTEGER GAR(60),BETA,ZTANK,AGE,YEAR,DATE
    LOGICAL DISJ,IMPL,STROKE,EQUIV,MODAL
    DOUBLE RL,MASS(10,10)
```

5.2 DIMENSION DECLARATION

A subscripted variable represents an element of an array of variables. Storage is reserved for arrays by the nonexecutable statements DIMENSION, COMMON, or a type statement.

5.2.1 CONSTANT DIMENSIONS

The standard form of the DIMENSION declaration 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 constants or variables (see paragraph 5.2.2).

Example:

```
DIMENSION A(10),B(20,3)
```

The values given by the subscript in the DIMENSION define the maximum value which a subscript may assume in a subsequent array reference.

The number of computer words reserved for an array is determined by the product of the subscripts in the subscript string and the type of the variable. A maximum of $2^{17}-1$ elements may be reserved in any one array. If the maximum is exceeded, a diagnostic is provided at compile time if constant subscripts are used.

Format:

```
COMPLEX ATOM
DIMENSION ATOM (10,20)
```

In the above declarations, the number of elements in the array ATOM is 200. Two words are used to contain a complex element; therefore, the number of computer words reserved is 400. This is also true for double precision arrays. For real, logical, and integer arrays, the number of words in an array equals the number of elements in the array.

If an array is dimensioned in more than one declaration statement, the first declaration holds and an informative diagnostic is provided.

Examples:

```
DIMENSION A(20,2,5)
DIMENSION MATRIX(10,10,10), VECTOR(100), ARRAY(16,27)
```

The maximum value a subscript may attain is as follows:

Dimension	Subscript Dimension	Subscript	Subscript Value	Maximum Subscript Value
1	(IDA)	(I)	(I)	IDA
2	(IDA, JDA)	(I, J)	(I+IDA*(J-1))	IDA*JDA
3	(IDA, JDA, KDA)	(I, J, K)	(I+IDA*(J-1) +IDA*JDA *(K-1))	IDA*JDA*KDA

I, J, and K are subscript expressions. IDA, JDA, and KDA are dimensions, for example, A(IDA, JDA, KDA).

5.2.2 VARIABLE DIMENSIONS

When an array identifier and some or all 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. Dimensions must not exceed the maximum array size specified by the DIMENSION declaration in the calling program.

Example:

```
SUBROUTINE X(A,L,M)
DIMENSION A(L,10,M)
```

5.3 COMMON DECLARATION

The COMMON declaration provides blocks of storage that can be referenced by more than one subprogram. The declaration reserves blank, numbered, and labeled blocks. Starting addresses for these blocks are indicated on the core map.

Areas of common information may be specified by the declaration:

COMMON/i_1/$list_1$/i_2/$list_2$...

The common block identifier, i, may be 1-7 characters. If the first character is alphabetic, the identifier denotes a labeled common block; remaining characters may be alphabetic or numeric. If the first character is numeric, 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 identifier. Labeled and numbered COMMON are treated identically by the compiler.

ANSI FORTRAN, X3.9-1966, specifies common block identifiers as consisting of up to six characters, of which the first is alphabetic.

Example:

```
COMMON/200/A,B,C
```

The following are common identifiers:

Labeled	Numbered
AZ13	1
MAXIM	146
Z	6600
XRAY	0

A common statement without a label or with blanks between the separating slashes is treated as a blank common block.

Example:

```
COMMON / / A,B,C     or     COMMON X,Y,Z(5)
```

List$_i$ is a string of identifiers representing simple and subscripted variables; formal parameters are not allowed. If a non-subscripted array name appears in the list, the dimensions must be defined by a type or DIMENSION declaration in that program. If an array is dimensioned in more than one declaration, a compiler diagnostic is issued. The order of simple variables or array storage within a common block is determined by the sequence in which the variables appear in the COMMON statements.

Labeled and numbered common blocks may be preset; data stored in them by DATA declarations is made available to any subprogram using the appropriate block. Data may not be entered into blank common blocks by the DATA declaration.

Examples:

```
COMMON/BLK/A(3)
DATA A/1.,2.,3./
COMMON/100/I(4)
DATA I/4,5,6,7/
```

COMMON is nonexecutable. Any number of blank COMMON declarations may appear in a program. If DIMENSION, COMMON, or type declarations appear together, the order is immaterial.

Since labeled and numbered common block identifiers are used only within the compiler, they may be used elsewhere in the program as other kinds of identifiers except subroutine or function names in the same job. A list identifier in one common block may not appear in another common block. (If it does, the name is doubly defined.)

At the beginning of program execution, the contents of all common areas are unpredictable except labeled common areas specified in a DATA declaration.

Examples:

```
COMMON A,B,C          }
                      }   Blank Common
COMMON/ /E,F,G,II     }
COMMON/BLOCKA/A1(15),B1,C1/BLOCKD/DEL(5,2),ECHO
COMMON/VECTOR/VECTOR(5),HECTOR,NECTOR
COMMON/9999/AX,BX,CX
```

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

Example:

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

If there is more than one COMMON statement in a program or subprogram with the same block name, all the elements are linked and stored consecutively as a single block.

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

```
COMMON/SUM/A,B,C,D          (main program)
COMMON/SUM/E(3),D           (subprogram)
```

In the above example, only the variable D is used in the subprogram. The unused variable E is necessary to space over the area reserved by A,B, and C.

Each subprogram using a common block assigns the allocation of words in the block. The identifiers used within the block may differ as to name, type, and number of elements; but the block identifier must remain the same.

Example:

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

The length of the block named TEST is 40 computer words. The length of the block numbered 36 is 3 computer words.

The subprogram may rearrange the allocation of words as in:

```
    SUBROUTINE ONE
    COMMON/TEST/A(10),G(10),K(10)
    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 are treated as floating point quantities; elements of K are treated as integer quantities.

The length of a common block other than blank common must not be increased by subprograms using the block unless that subprogram is loaded first by the SCOPE loader. The symbolic names used within the block may differ, however, as shown above.

5.4 EQUIVALENCE DECLARATION

The EQUIVALENCE declaration 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 subscripted variable names; formal arguments are not allowed. Subscripts may only be integer constants. 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+(j-1)*I+(k-1)*I*J))*E)

where E is 1 or 2 depending on A's word length, 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) can be represented by A(7).

EQUIVALENCE is most commonly used when two or more arrays can share the same storage locations. The lengths need not be equal.

Example:

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

The EQUIVALENCE declaration 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 since the values of array I are read into the storage locations previously occupied by A.

Variables requiring two memory positions which appear in EQUIVALENCE statements must be declared to be COMPLEX or DOUBLE prior to their appearance in such statements.

ANSI FORTRAN, X3.9-1966, does not require type declaration prior to equivalence.

Example:

```
      COMPLEX DAT,BAT
      DIMENSION DAT(10,10),BAT(10, 10),CAT(10,10)
      DOUBLE PRECISION CAT
      COMMON/IFAT/FAT(2,2)
      EQUIVALENCE (DAT(6,3),FAT(2,2) ),(CAT,BAT)
      .
      .
      .
```

EQUIVALENCE is nonexecutable and can appear anywhere in the program or subprogram. However, if it appears after the first executable statement, an informative diagnostic is provided.

Any variable may be made equivalent to any other variable, provided that no two variables in any one group are in COMMON. The variables may be with or without subscripts.

ANSI FORTRAN, X3.9-1966, does not specify that variables may be without subscript.

The following examples illustrate changes in block lengths caused by the EQUIVALENCE declaration:

Given:

Arrays A and B
Sa subscript of A
Sb subscript of B

Examples:

A and C in common, B not in common

$Sb \leq Sa$ is a permissible subscript arrangement

$Sb > Sa$ is not

The design of this compiler prevents the following use of EQUIVALENCE.

```
                        Block 1

origin     A(1)                     COMMON/1/A(4),C
           A(2)      B(1)           DIMENSION B(5)
           A(3)      B(2)           EQUIVALENCE (A(3),B(2))
           A(4)      B(3)
           C         B(4)
                     B(5)
           DIMENSION FAT (6)
           COMMON/COMA/SKINNY
           EQUIVALENCE (SKINNY,FAT(n))
```

The last statement will be flagged fatally if $n > 1$.

5.5 DATA DECLARATION

Values may be assigned to program variables or labeled common variables with the DATA declaration.

Format:

DATA $k_1,\ldots,k_n/d_1,j*d_2,\ldots,d_n/,k_1,\ldots,k_n/d_1,\ldots,d_n/,\ldots$

k_i Identifiers representing simple variables, array names, or variables with integer constant subscripts or integer variable subscripts (implied DO loop notation)

d_i Literals and signed or unsigned constants

j Integer constant repetition factor that causes the literal following the asterisk to be repeated k times. If k is non-integer, a compiler diagnostic occurs

A semicolon cannot be used in the character string of data entered under L, R, or H control.

DATA is nonexecutable and can appear anywhere in the program or subprogram. When DATA appears with DIMENSION, COMMON, EQUIVALENCE, or a type declaration, the statement that dimensions any arrays used inthe DATA statement must appear prior to the DATA statement. Variables in blank common or formal arguments may not be preset by a DATA declaration. The index variable in an implied DO loop in a DATA statement must be an implicit integer.

> ANSI FORTRAN, X3.9-1966, does not specify that presetting a labeled COMMON may be done other than in a BLOCK DATA subprogram.

Only single-subscript, DO-loop-implying notation is permissible. This notation may be used for storing constant values in arrays.

> ANSI FORTRAN, X3.9-1966, does not specify the use of DO-loop-implying notation for storing constants in arrays.

Examples:

1.
```
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
```

2.
```
DIMENSION TWO(2,2)

DATA TWO(1,1),TWO(1,2),TWO(2,2),TWO(2,1)/1.,2.,3.,4./

     Array TWO:   TWO(1,1)     1.
                  TWO(2,1)     4.
                  TWO(1,2)     2.
                  TWO(2,2)     3.
```

```
3.  DIMENSION SINGLE(3,2)

    DATA (SINGLE(I),I=1,6)/1.,2.,3.,1.,2.,3./

         Array SINGLE:  SINGLE(1,1)   1.
                        SINGLE(2,1)   2.
                        SINGLE(3,1)   3.
                        SINGLE(1,2)   1.
                        SINGLE(2,2)   2.
                        SINGLE(3,2)   3.
```

In the DATA declaration, the type of the constant stored is determined by the structure of the constant rather than by the variable type 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 symbolic name A. Data types requiring two words per element must be properly specified to maintain correct correspondence in memory.

There should be a one-to-one correspondence between the variable names and the list. This is particularly important in arrays in labeled and numbered common.

```
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 discarded; B is unmodified and an error is issued. 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 content of C(3), that is, location C+2, is not defined.

When the number of list elements exceeds the range of the implied DO, the excess list elements are not stored, and a diagnostic is issued.

```
DATA (A(I),I=1,5,1)/1.,2.,...,10./
```

The excess values 6. through 10. are discarded.

Examples:

```
1.  DATA LEDA,CASTOR,POLLUX/15,16.0,84.0/

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

2.  DATA A(1,3)/16.239/

        ARRAY A

            A(1,3)                16.239

3.  DIMENSION B(10)

        DATA B/O000077, O000064,3*O000005, 5*O000200/

            ARRAY B               O77
                                  O64
                                  O5
                                  O5
                                  O5
                                  O200
                                  O200
                                  O200
                                  O200
                                  O200

4.  COMMON/HERA/C(4)

        DATA C/3.6,3*10.5/

            ARRAY C                3.6
                                  10.5
                                  10.5
                                  10.5
```

5.
```
COMPLEX PROTER (4)

     DATA PROTER/4*(1.0,2.0)/

          ARRAY PROTER          1.0
                                2.0
                                1.0
                                2.0
                                1.0
                                2.0
                                1.0
                                2.0
```

6.
```
DIMENSION MESSAGE (3)

     DATA MESSAGE/9HSTATEMENT,2HIS,10HINCOMPLETE/

          ARRAY MESSAGE                   STATEMENT IS
                                          INCOMPLETE
```

Data declaration statements of the following forms may also be used to assign constant values to program or common variables at load time.

DATA(i_1=value list), (i_2=value list), . . .

The variable identifier, i, may be:

non-subscripted variable

array variable with constant subscripts

array name (implying the whole array)

The value list is either a single constant or set of constants whose number is equal to the number of elements in the named array.

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 is provided.

ANSI FORTRAN, X3.9-1966, does not specify the preceding form of the DATA statement.

Examples:

```
COMMON/DATA/GIB(10)
DATA ( (GIB(I),I=1,10)=1.,2.,3.,7(4.32) )
COMMON/DATA/ROBIN(5,5,5)
DATA (ROBIN(4,3,2)=16.)
```

5.5.1 BLOCK DATA SUBPROGRAM

A block data subprogram may be used to enter data into labeled or numbered common prior to program execution in place of a DATA declaration and it may appear more than once in a FORTRAN program. However, blank common variables cannot be preset.

The form of a BLOCK DATA subprogram is:

```
BLOCK DATA n
      .
      .
      .
FORTRAN declaration statements only
      .
      .
      .
END
```

Where n is blank or any acceptable alphanumeric identification of seven characters or less, all elements in the common blocks must appear in a COMMON declaration in the subprogram even if they are not in the DATA declaration.

ANSI FORTRAN, X3.9-1966, does not specify a non-blank identifier for BLOCK DATA.

Examples:

```
BLOCK DATA
COMMON/ABC/A(5),B,C/DEF/D,E,F
COMPLEX D,E
DOUBLE PRECISION F
DATA (A(L),L=1,5)/2.3,3.4,3*7.1/,B/2034.756/,D,E,F/2*(1.0,2.5),17.86972415872D30/
END
```

5.6 LEVEL DECLARATION

The LEVEL declaration provides a means of allocating and referencing data within a generalized storage hierarchy.

The form of the LEVEL declaration is:

```
LEVEL n, list
```

n is an unsigned integer representing the storage level to which the variables and arrays given in list are to be allocated. Permissible level numbers and the areas to which they refer are:

1	small core memory
2	large core memory, direct access
3	large core memory, block transfer

list consists of one or more variables and/or array names, separated by commas. Dimensioning and type information is not given in a LVEL statement. Data assigned to levels 2 and 3 must also appear in COMMON statements or as dummy arguments in SUBROUTINE statements. When a large core memory variable does not appear in a COMMON statement, but instead, is equivalent to a large core memory variable which is in COMMON, then the message LEVEL 2 OR 3 VARIABLE NOT IN COMMON is printed. This message is a warning only and production of a relocatable program is not inhibited.

A LEVEL statement, when present, must precede the first executable statement of a program or subprogram.

Variables and arrays appearing in LEVEL statements can also appear in DATA, DIMENSION, EQUIVALENCE, COMMON, type, SUBROUTINE, and FUNCTION statements.

No restrictions are placed in the manner in which variables or arrays allocated to levels 1 and 2 may be referenced with the exceptions of the statement FUNCTION, its references, and variable FORMAT specifications. Data assigned to level 3 can be referenced only in COMMON, CALL SUBROUTINE, FUNCTION, TYPE, and DIMENSION statements. To store or retrieve LEVEL 3 variables, use the WRITECor READEC system subroutines, respectively. Refer to Appendix K for detail or CALL WRITEC and CALL READEC. The CALL READEC or CALL WRITEC data transfer operations are the only operations that can be performed on level 3 data. Data allocated to levels 2 and 3 cannot be utilized as actual or formal parameters of statement functions.

If a large core variable appears as an actual argument in a reference to a FUNCTION subprogram or to a library function subroutine, instructions are compiled for transferring the large core variable to a small core memory location; the small core memory address is transmitted as the argument to the FUNCTION subprogram or library function subroutine. This implies that the actual large core argument may not specify a result location or an array name to the FUNCTION subprogram or library function subroutine. Variable FORMAT specifications must not reside in large core memory.

An informative diagnostic will be issued if the level of any variable is multiply defined and the level first declared will be assumed.

All members of a common block must be allocated to the same level of storage. A fatal diagnostic will be issued if conflicting levels are declared for different members of a common block. An informative diagnostic will be issued if some but not all members of a common block are declared in LEVEL statement(s) and all members will be assigned to the declared level.

If a variable or array name appears as an actual parameter in a CALL statement the corresponding formal parameter in the called subprogram must appear in a LEVEL statement in the called subprogram and must be allocated to the same level as the actual parameter.

If a variable or array name appears in a LEVEL statement and in an EQUIVALENCE statement, the equivalenced variables must all be allocated to the same level.

Names of variables and arrays not included in a LEVEL statement will be allocated to level 1 (SCM) by default.

> ANSI FORTRAN, X3.9-1966, does not specify a LEVEL statement.

The following example illustrates the use of levels 2 and 3 and demonstrates their differences:

```
                  PROGRAM LTEST (OUTPUT)
                  LEVEL 2, B
                  LEVEL 3, C
                  DIMENSION A(10), B(10), C(10), D(10)
                  COMMON /BB/B
                  COMMON /CC/C
                  DO 1 I=1,10
10                A(I) =1.0
11        1       D(I) =2.0
12                DO 2 I=1,10
16        2       B(I) = A(I)
     C               LEVEL 2 IS USED FOR DIRECT ACCESS OF LARGE CORE MEMORY
20                PRINT 5, (A(I),B(I),D(I), I=1,10)
          5       FORMAT (1H ,2HA ,F10.1,5X,2HB ,F10.1,5X,2HD ,F10.1/)
42                CALL WRITEC (A,C,10)
     C               LEVEL 3 IS USED FOR BLOCK TRANSFERS, TO AND FROM LARGE
                     CORE MEMORY
46                CALL READEC (D,C,10)
52                PRINT 6, (D(I), I=1,10)
          6       FORMAT (1H ,2X,2HD ,F10.1,5X,2HD ,F10.1/)
60                END
```

```
     A        1.0        B       1.0     D     2.0 ⎫
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎬      Level 2
     A        1.0        B       1.0     D     2.0 ⎪      statement output
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎪
     A        1.0        B       1.0     D     2.0 ⎭
        D        1.0      D        1.0 ⎫
        D        1.0      D        1.0 ⎪
        D        1.0      D        1.0 ⎬   Level 3 statement output
        D        1.0      D        1.0 ⎪
        D        1.0      D        1.0 ⎭
```

CONTROL STATEMENTS 6

Program execution normally proceeds from statement to statement as they appear in a 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. A transfer to a nonexecutable statement will result in a program error, which is always recognized during compilation. With the DO statement, a predetermined sequence of instructions can be repeated any number of times by incrementing a simple integer variable after each iteration.

6.1 GO TO STATEMENT

GO TO statements transfer control to a labeled statement whose reference is fixed or which is selected during execution of the program. The statement labels used in the GO TO statements must be associated with executable statements in the same program unit as the GO TO statement.

6.1.1 UNCONDITIONAL GO TO

Format:

```
    GO TO k
```

k is a statement label and remains constant.

Execution of this statement discontinues the current sequence of execution and resumes execution at the statement labeled k.

Example:

```
        GO TO 10
  5     DIF = DIF - SUM
  10    SUM = SUM + 1
```

Statement 5 is skipped during execution of this sequence.

6.1.2 ASSIGNED GO TO

Format:

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

GO TO m

ANSI FORTRAN, X3.9-1966, does not specify GO TO n.

This statement acts in association with an ASSIGN statement as a variable-branched GO TO. m is a simple integer variable assigned an integer value n in a preceding ASSIGN statement (paragraph 6.1.3). The n_i are statement labels. As shown, the parenthetical statement label list need not be present.

Once having been defined by an ASSIGN statement, the variable m may not be referenced by any statement other than GO TO m until it is redefined.

The comma after m is optional. However, when the list is omitted, the comma must be omitted. m cannot be defined as the result of a computation. No compiler diagnostic is given if m is computed, but the object code is incorrect. If an assignment has not been made for an assigned GO TO statement and m is equal to zero, a diagnostic is provided at object time. If m is non-zero, a valid assignment is assumed. FORTRAN does not preset all locations to zero, nor does it check all the possible statement labels that may be assigned.

ANSI FORTRAN, X3.9-1966, requires the comma.

Example:

```
       ASSIGN 15 to K
       GO TO 60
  15   J = 9
       L = (I**2) + J
 100   ASSIGN 20 to K
       GO TO 60
         .
         .
         .
  60   CONTINUE
         .
         .
         .
       GO TO K
  20   CONTINUE
```

When the program executes the ASSIGN statement, K has the value 15.

Control moves to the next statement which causes a jump to statement 60, CONTINUE.

The program executes the statements following 60 in sequence until it reaches GO TO K. Since K previously has been assigned the value 15, control jumps to statement 15. Statement 15 equates J to the value 9. The following statement uses this value of J in an arithmetic expression.

In the next statement, the program assigns 20 to K. The next step causes a jump to 60, a CONTINUE. The program goes through the steps following 60 until it reaches GO TO K. As K has been assigned the value 20, control jumps to statement 20, CONTINUE, and proceeds in sequence.

6.1.3 ASSIGN STATEMENT

Format:

```
ASSIGN k TO m
```

k is one of the statement labels appearing in the GO TO list; m is the simple integer variable in the assigned GO TO statement. At the time of execution of an assigned GO TO statement, the current value of m must have been assigned by an ASSIGN statement.

Example:

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

Statement number 10 will be executed next.

6.1.4 COMPUTED GO TO

Format:

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

This statement acts as a many-branch GO TO; i is present or computed prior to its use in the GO TO.

The n_i are statement labels and i is a simple integer variable. If $i < 1$ or if $i > m$, the transfer is undefined and an object time diagnostic will be issued indicating the point at which the error was detected. If $1 \leq i \leq m$, the transfer is to n_i.

The comma separating the statement number list and the index is optional.

ANSI FORTRAN, X3.9-1966, requires the comma.

Example:

```
N=3
.
.
.
GO TO (100, 101,102,103) N
```

Statement number 102 will be the selected control transfer.

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

Example:

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

Control transfers to statement 21.

6.2 IF STATEMENT

The IF statement is used to transfer control conditionally, in accordance with a logical determination. At time of execution, an expression in the IF statement is evaluated and the result determines the statement to which the jump will be made.

6.2.1 THREE-BRANCH ARITHMETIC IF

The form of the three-branch arithmetic IF is:

IF (c)n_1, n_2, n_3

c is an arithmetic expression, and the n_i are statement labels. This statement tests the evaluated expression c and jumps accordingly as follows:

$c < 0$	jump to statement n_1
$c = 0$	jump to statement n_2
$c > 0$	jump to statement n_3

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

ANSI FORTRAN, X3.9-1966, does not specify an IF statement with complex c.

Example:

```
    IF (IOTA-6)3, 6, 9
```

If the evaluation of the expression IOTA-6 produces a negative result, control transfers to the statement labeled 3; if zero, to 6; if positive, to 9.

6.2.2 TWO-BRANCH ARITHMETIC IF

The second form of the arithmetic IF statement, the two-branch IF, is allowed.

Format:

IF (e)n_1, n_2

e is either a masking or an arithmetic expression; e is evaluated and control is transferred as follows:

$e \neq 0$ jump to statement n_1

$e = 0$ jump to statement n_2

Example:

```
         IF (I*J*DATA(K))100, 101
100      IF (I*Y*K)105, 106
```

ANSI FORTRAN, X3.9-1966, does not specify the two-branch arithmetic IF.

6.2.3 ONE-BRANCH LOGICAL IF

The form of the one-branch logical IF is:

IF (ℓ)s

ℓ is a logical or relational expression and s is any executable statement except another logical IF, a DO statement,or an END. If ℓ is true (not plus zero), the statement s is executed. If ℓ is false (plus zero),the statement immediately following the IF statement is executed.

Example:

```
IF (A.LE.2.5) A = 2.0
```

When this statement is executed, the value of A will be compared with 2.5. If it is less than or equal to 2.5, A will be set to the value 2.0. If the comparison shows A to be greater than 2.5, control will proceed to the statement following.

6.2.4 TWO-BRANCH LOGICAL IF

The form of the two-branch logical IF is:

IF (ℓ)n_1, n_2

ℓ is a logical or relational expression and the n_i are statement labels.

The evaluated expression is tested for true (not plus zero) or false (plus zero) condition. If ℓ is true, the jump is to statement n_1. If ℓ is false, the jump is to statement n_2.

ANSI FORTRAN, X3.9-1966, does not specify the two-branch logical IF.

Example:

IF (ℓ) 5, 6

At the time of execution, ℓ is tested for true or false condition. If true, control transfers to statement 5. If false, control transfers to statement 6.

6.3 DO STATEMENT

A DO statement makes it possible to repeat a group of statements a designated number of times using an integer variable whose value is progressively altered with each repetition. The initial value, final value, and rate of increase of this integer variable is defined by a set of indexing parameters included in the DO statement. The range of the repetitions extends from the DO statement to the terminal statement and is called the DO loop.

The form of a DO statement is:

DO n i = m_1, m_2

Do n i = m_1, m_2, m_3

n — Label of the terminal statement of the loop.

i — Simple integer variable called the index variable. With each repetition, its value is altered progressively by the increment parameter m_3. Upon exiting from the range of a DO, the control variable remains defined as the last value acquired in execution of the DO if the exit results from execution of a GO TO or IF only. If the exit results from the DO loop being satisfied, the index variable is no longer well defined.

m_1 — Initial parameter, the value of i at the beginning of the first loop.

m_2 — Terminal parameter. When the value of i surpasses the value of m_2, DO execution is terminated and control goes to the statement immediately following the terminal statement.

m_3 — Increment parameter, the amount i is increased with each repetition. If it has the value 1, it may be omitted (first form above).

The DO statement, the statement labeled n, and any intermediate statements constitute a DO loop. The first statement in the range of the DO must be an executable statement. The terminal statement can not be a terminal, GO TO of any form, arithmetic IF, two-branch logical IF, RETURN, STOP, PAUSE, another DO statement, a nonexecutable statement, or a logical IF containing any of these forms.

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 cause a diagnostic. None of the indexing parameters may exceed 2^{17}-2.

The values of m_1, m_2, and m_3 may be changed during the execution of the DO loop.

The initial and terminal parameters (m_1, m_2), if variable, may assume positive or negative values, or zero. The increment parameter (m_3) must be greater than zero.

ANSI FORTRAN, X3.9-1966, states that at time of execution of the DO, m_1, m_2, and m_3 must be greater than zero.

Example:

```
        DO 25 I=1,100
    25  A(I)=A(I)+B(I)
```

The index variable I is incremented by one for each cycle until the DO loop is executed 100 times. The control is then transferred to the statement immediately following statement 25.

```
        DO 12 I=1,10,2
        J=I+K
        X(J)=Y(J)
    12  CONTINUE
```

I is set to the initial value of one and incremented by two on each of the following cycles. When the execution of the fifth cycle (I=9) is completed, control passes out of the DO loop.

The following program calculates the sum of all odd numbers and the sum of all even numbers in the range of 1 to 100.

Format:

```
        IODD=0
        IEVEN=0
        DO 25 I=1,99,2
        IODD=IODD + I
        J=I + 1
    25  IEVEN=IEVEN+J
```

The first two steps zero out the counters for the odd and the even numbers. The DO statement initiates a loop that begins at the index value of 1 and increments in steps of 2. This series provides the odd numbers. The J = I + 1 statement provides the series of even numbers by adding a 1 to each of these values. The operation of this DO loop is tabulated in Table 6-1.

TABLE 6-1. DO LOOP OPERATION CHART

Loop	I	IODD=IODD+I	(store)	J=I+1	(store)	IEVEN=IEVEN+J	(store)
1	1	1=0+1	(1)	2=1+1	(2)	2=0+2	(2)
2	3	4=1+3	(4)	4=3+1	(4)	6=2+4	(6)
3	5	9=4+5	(9)	6=5+1	(6)	12=6+6	(12)
4	7	16=9+7	(16)	8=7+1	(8)	20=12+8	(20)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	Successive values of control variable I which is the sequence of odd numbers		Progressive addition of odd numbers		Sequence of even numbers/		Progressive addition of even numbers

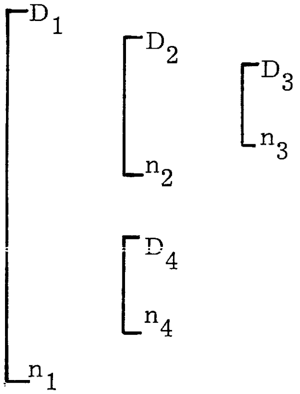
6.3.1 DO LOOP EXECUTION

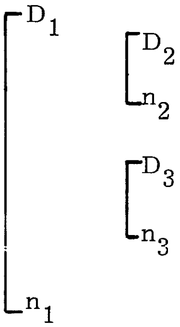
The initial value of i, m_1, is increased by m_3 and compared with m_2 after executing the DO loop once. If i does not exceed m_2, the loop is executed a second time. Then, i is again increased by m_3 and again compared with m_2. This process continues until i exceeds m_2. Control passes to the statement immediately following n and the DO loop is satisfied.

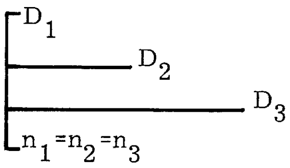
Should m_1 exceed m_2 on the initial entry to the loop, the loop is executed once and control is passed to the statement following n. When the DO loop is satisfied, the index variable i is no longer well defined. If a transfer out of the DO loop occurs before the DO is satisfied, the value of i is preserved and may be used in subsequent statements.

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
     .
     .
     .
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     A=B*C
```

Example:

To test Fermat's Last Theorem with combinations of integer values up to 1000, the theorem states that the equation

$$X^n + Y^n = Z^n$$

is not valid for positive integer values of X, Y, and Z when n is an integer greater than 2.

Letting I, J, K equal X, Y, Z to imply integer values, the test may be programmed as follows:

```
        PROGRAM FERMAT
        DO 13 N = 3, 1000
        DO 13 I = 1, 1000
        DO 13 J = 1, 1000
        DO 13 K = 1, 1000
        IF (I**N+J**N-K**N)13, 7, 13
7       WRITE (3, 100)I, J, K, N
100     FORMAT(6HEUREKA/4I5)
13      CONTINUE
        STOP
        END
```

Example:

A loan is repaid in N monthly payments with each payment equal to P and with an interest rate of R. The total repaid, S, is given by:

$$S = \frac{P}{R} \left(1- \frac{1}{(1+R)} N \right)$$

The following program calculates the sums repaid for monthly payments of 24, 30, and 36 months in amounts of 20, 30, 40 and 50 dollars at interest rates of .06, .07, .08, .09, and .10.

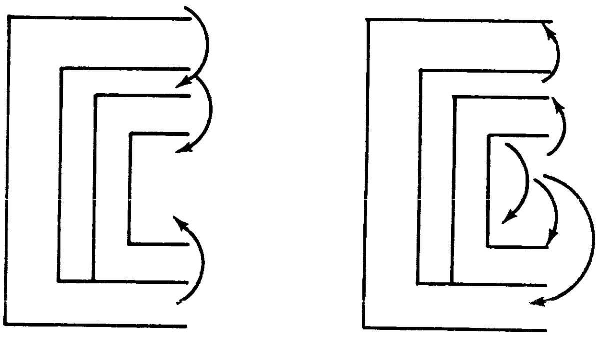
```
        DIMENSION SUM (5)
        DO 30 N = 24, 36, 6
        DO 20 J = 20, 50, 10
        DO 10 I = 6, 10
        R = I*0.01
10      SUM (I) = J/R*(1.-1./((1.+R)**N))
20      WRITE (3,40) (SUM(K),K=1,5)
40      FORMAT (5F10.2)
30      CONTINUE
```

This would print out the sums, 5 to a line, according to the five interest rates. The following tabulation of the printed output shows how the cycling proceeds through the DO loops with the innermost loop varying the most rapidly and the outermost loop varying the least rapidly.

Months	Amount	Rate	Months	Amount	Rate
24	20	.06	30	40	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
	30	.06		50	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
	40	.06	36	20	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
	50	.06		30	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
30	20	.06		40	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10
	30	.06		50	.06
		.07			.07
		.08			.08
		.09			.09
		.10			.10

6.3.2 DO LOOP TRANSFER

In a DO nest, a transfer may be made from one DO loop into a DO loop that contains it. The transfer should not be made from the outer DO loop to the inner DO loop without first executing the DO statement of the inner DO loop.



Not Allowed

Allowed

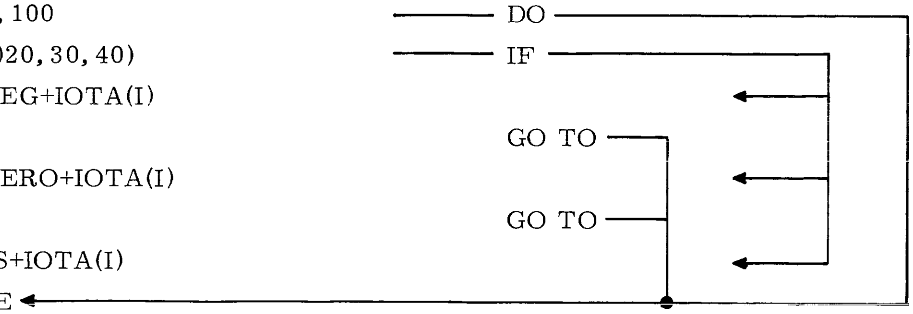
A DO is said to have an extended range if, once the DO statement has been executed and before the loop is satisfied, control is transferred out of the DO range to perform some calculation and then transferred into the range of the DO. The return must not be made to the terminal statement.

When a statement is the terminal statement of more than one DO loop, the label of that terminal statement may not be used in any GO TO or arithmetic IF statement in the nest, except in the range of the innermost DO.

Test 100 values for sign and accumulate three sums: one for negative values, one for zero, and one for positive.

Format:

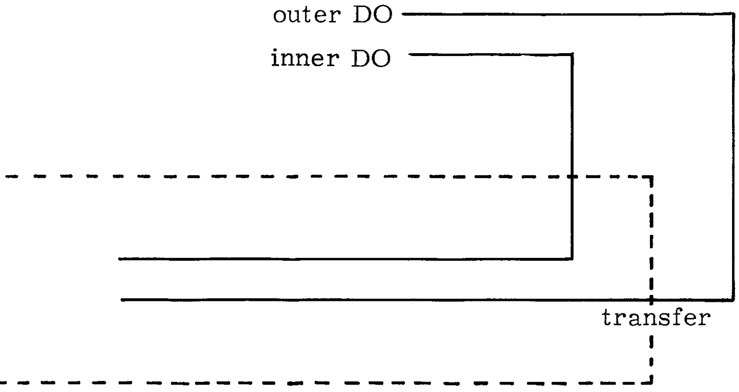
```
      PROGRAM TEST
      DIMENSION IOTA (100)
      READ (1,10) (IOTA(I),I=1,100)
 10   FORMAT(10I5)
      INEG = 0
      IZERO = 0
      IPOS = 0
      DO 50 I=1,100
      IF(IOTA(I)20,30,40)
 20   INEG = INEG+IOTA(I)
      GO TO 50
 30   IZERO=IZERO+IOTA(I)
      GO TO 50
 40   IPOS=IPOS+IOTA(I)
 50   CONTINUE
```



Compare two arrays of numbers and print out all sets of equivalent values.

Format:

```
         DO 40 I = 1,20
         DO 30 J = 1,20
         K = I
         L = J
         IF(A(I).EQ.B(J)) GO TO 50
   25    L=J+I
   30    CONTINUE
   40    CONTINUE
         STOP
   50    WRITE(3,10) A(K),B(L)
         GO TO 25
   10    FORMAT(F8.5,3H=,F8.5)
```



Control can be transferred out of a DO loop or nest of DO loops and returned, provided the indexing parameters are not altered and control is transferred back to the range of the same DO loop from which the exit was made.

Examples:

1. This example is acceptable since the statement GO TO 2 occurs from the innermost DO loop.

```
1   GO TO 3
2   A(I)=A(I)+B(I,J)
    GO TO 1
3   DO 1 I=1,M
    A(I)=0
    DO 1 J=1,N
    GO TO 2
1   CONTINUE
```

2. This example is not acceptable since the statement:

    IF(A(I).NE.0.0) GO TO 100

   does not occur from the innermost DO loop.

```
    DO 100 I=1,N
    IF(A(I).NE.0.0) GO TO 100
    DO 100 J=1,J
    B(I,J)=B(I,J)/A(I)
100 CONTINUE
```

Statement number 100 causes index to increment for the inner DO loop first then for the outer DO loop.

3. This example is acceptable since statement number 3 is in the range of the DO for I index and not in the range of the DO for J index.

```
4   IF(A(I)3,5,5)
    DO 3 I=1,M
    GO TO 4
5   DO 2 J=1,N
1   A(I)=A(I)+B(I,J)
2   CONTINUE
3   CONTINUE
```

For the preceding examples, the terminal statement number of the DO loop must be referenced prior to the DO statement as a later reference to such a statement number produces a message indicating a missing statement number.

6.4 CONTINUE STATEMENT

CONTINUE

The CONTINUE statement is most frequently used as the last statement of a DO loop to provide a loop termination. 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.

An example where a CONTINUE statement is needed is:

```
    DO 10 I=1,N
    IF (A(I).EQ.B) GO TO 20
10  CONTINUE
```

6.5 PAUSE STATEMENT

PAUSE

PAUSE n

n ≤ 5 octal digits without an O prefix or B suffix. PAUSE n rolls out the program and requests operator action at the station submitting the job. The words PAUSE n are displayed as a dayfile message. An operator entry from the console can continue or terminate the program. Program continuation proceeds with the statement immediately following PAUSE. If n is omitted, it is understood to be blank.

In general, the PAUSE statement is no longer used because the FORTRAN programmer does not directly oversee the running of this program.

6.6 STOP STATEMENT

STOP

STOP n

n ≤ 5 octal digits without an O prefix or B suffix. STOP or STOP n terminates the program execution and returns control to the operating system. If n is omitted, it is understood to be blank. Common usage of STOP n has been to alert the program that his program has abnormally or specifically ended.

6.7 RETURN STATEMENT

RETURN

A procedure subprogram must contain one or more RETURN statements to indicate the end of logic flow within the subprogram and return control to the calling program. It omitted, the successful execution of that subprogram will terminate the entire program.

In function subprograms, control returns to the statement containing the function reference and implies that the value represented is the name of the functional availability. In a subroutine subprogram, control returns to the next executable statement following the CALL. A RETURN statement in the main program causes an exit to the operating system.

6.8 END STATEMENT

END

END must be the final statement in a program or subprogram. It is executable in the sense that it effects termination of the program. The END statement may not be labeled.

The END statement may include the name of the program or subprogram which it terminates; however, any information appended to the END statement is ignored by the compiler.

ANSI FORTRAN, X3.9-1966, does not allow END as the last executable statement. Also, it must contain the character END only and must not be continued on another line.

PROGRAM, PROCEDURES AND SUBPROGRAMS 7

7.1 SOURCE PROGRAM

A source program consists of a main program and optionally one or more auxiliary procedures and subprograms. The subprograms can be compiled separately and combined with the main program for execution.

7.2 MAIN PROGRAM

The first statement of a main program should contain the name as an alphanumeric identifier of 1-7 characters. The parameter list is optional on all forms. If the first card of a program is not one of the following forms, a PROGRAM with a blank name and I/O files are assumed. If more files than I/O are necessary, a PROGRAM card is required.

Format:

PROGRAM name($f_1,\ldots,f_n$)

FORTRAN II PROGRAM name($f_1,\ldots,f_n$)

ANSI FORTRAN, X3.9-1966, does not specify the PROGRAM statement.

The f_i represent the names of all I/O files required by the main program and its subprograms. n must not exceed 30. These parameters may be changed at execution time. At compile time, they must satisfy the following conditions.

1. The file name INPUT (references standard input unit) must appear if any READ statement is included in the program or its subprograms.

2. The file name OUTPUT (references standard output unit) must appear if any PRINT statement is included in the program or its subprograms. OUTPUT is required for obtaining a listing of execution diagnostics.

3. The file name PUNCH must appear if any PUNCH statement is included in the program or its subprograms.

4. The file name TAPE i, must appear if a READ (i,n), WRITE (i,n), READ (i), or WRITE (i) statement is included in the program or its subprogram (i is defined in Chapter 10).

5. If I is an integer variable name for a READ (I,n) WRITE (I,n), READ (I), or WRITE (I) statement which appears in the program or its subprogram, the file names TAPE $i_1,\ldots,$TAPE i_k must appear. The integers $i_1,\ldots,i_k$ must include all values which are assumed by the variable I. The file name TAPE I may not appear in the list of arguments to the main program.

File names may be made equivalent at compile time. A PROGRAM statement having specified buffer lengths will be accepted, but the compiler will ignore them. (See Appendix E for details on file name handling at execution time.) In the list of parameters, equivalenced file names must follow those to which they are made equivalent. The equivalenced name appears on the left hand side of m= and the name to which it is made equivalent appears on the righthand side. Their corresponding parameter positions may not be changed at execution even though the names of the files to which they are made equivalent may be changed at that time.

Example:

```
    PROGRAM ORB (INPUT,OUTPUT,TAPE 1 = INPUT, TAPE 2 = OUTPUT)
```

All input normally provided by TAPE 1 would be extracted from INPUT and all listable output normally recorded on TAPE 2 would be transmitted to the OUTPUT file.

7.3 PROGRAM COMMUNICATION

The main program and subprograms communicate with each other via arguments and COMMON variables. Subprograms may call or be called by any other subprogram as long as the calls are nonrecursive. That is, if program C calls D, D may not call C. A calling program is a main program or subprogram that refers to another subprogram. A subroutine referenced by a program may not have the same name as the program.

7.4 SUBPROGRAM COMMUNICATION

Subprograms, functions, and subroutines use arguments as one means of communication. The arguments appearing in a subroutine call or a function reference are actual arguments. The corresponding arguments appearing with the program, subprogram, statement function, or library function name in the definition are formal arguments. One or more of the formal arguments or common variables can be used to return output to the calling program.

7.5 PROCEDURES AND SUBPROGRAMS

A FORTRAN program consists of a main program with or without auxiliary procedures and subprograms. Auxiliary sets of statements are used to evaluate frequently used mathematical functions, to perform repetitious calculations, and to supply data specifications and initial values to the main program. FORTRAN provides six such procedures and subprograms:

Statement function

Intrinsic function

Basic external function

External function

External subroutine

Block data subprogram

The intrinsic function and the basic external function are furnished with the system. They are used to evaluate standard mathematical functions. The others are user-defined. The statement function and intrinsic function are compiled within the main program or subprogram, the basic external function is furnished with the system, and the others are compiled separately. The first five are referred to as procedures since each is an executable unit that performs its set of calculations when referenced. The first four are called functions. They return a single result to the point of reference. The last three subprograms are user-defined and are compiled independently. The block data subprogram supplies specifications and initial values to the main program. Table 7-1 outlines these categorical divisions.

The use of procedures and subprograms is determined by their particular capabilities and the needs of the program being written. If the program requires the evaluation of a standard mathematical function, an intrinsic function or a basic external function is used (Appendix C). If a single non-standard computation is needed repeatedly, a statement function may be inserted in the program. If a number of calculations are required to obtain a single result, a function subprogram may be written. If a number of calculations are required to obtain an array of values, a subroutine can be written. When the program requires initial values, a BLOCK DATA subprogram should be used.

7.5.1 PROCEDURE IDENTIFIERS

A procedure identifier is a symbolic name of up to seven alphanumeric characters, the first of which must be alphabetic.

FORTRAN, X3.9-1966, limits all symbolic names to six characters.

There is no type associated with a symbolic name that identifies a SUBROUTINE. For a function subprogram, type is specified either implicitly by its name, explicitly in the FUNCTION statement, or in a type statement. For a statement function, type is specified either implicitly by its name or explicitly in a type statement.

7.5.2 FORMAL ARGUMENTS

Formal arguments appear within the FUNCTION or SUBROUTINE statement or in the statement function definition and serve only to allocate data values in these auxiliary routines. For this reason, they are often referred to as dummy arguments.

Formal arguments may be the names of arrays, simple variables, library functions (basic external functions), and subprograms (FUNCTION and SUBROUTINE). Since formal arguments are local to the subprogram containing them, they may be the same as names appearing outside the procedure.

No element of a formal argument list may appear in an EQUIVALENCE, COMMON, or DATA statement within a subprogram. If it does, a compiler diagnostic results.

When a formal argument represents an array, it must be dimensioned within the subprogram. If it is not declared, the array name must appear without subscripts and only the first element of the array is available to the subprogram.

TABLE 7-1. SUBDIVISION OF PROCEDURES AND SUBPROGRAMS

<table>
<tr><th>Statement Function</th><th>Intrinsic Function</th><th>Basic External Function</th><th>External Function</th><th>External Subroutine</th><th>Block Data Subprogram</th></tr>
<tr><td>User-Defined</td><td colspan="2">Compiler-Defined</td><td colspan="3">User-Defined</td></tr>
<tr><td colspan="2">Compiled within the referencing program</td><td>Not Compiled
-LIBRARY-</td><td colspan="3">Compiled externally to the referencing program</td></tr>
<tr><td colspan="5">PROCEDURE: Any defined calculation that can be referenced and which will exchange values between reference and definition through a list of arguments.</td><td></td></tr>
<tr><td colspan="2"></td><td colspan="3">EXTERNAL PROCEDURE: A procedure that is defined externally to the program unit that references it.</td><td></td></tr>
<tr><td colspan="6">FUNCTION: A procedure that supplies a single result to be used at the point of reference. It can also modify the arguments.</td></tr>
<tr><td colspan="2"></td><td colspan="4">EXTERNAL FUNCTION: A function defined externally to the program unit that references it.</td></tr>
<tr><td colspan="3" rowspan="2"></td><td colspan="3">SUBPROGRAM: A user-defined set of statements compiled independently from the program unit which references it or to which it supplies specifications and initial values.</td></tr>
<tr><td colspan="2">PROCEDURE SUBPROGRAM: An external procedure that is defined by FORTRAN statements.</td><td>SPECIFICATION SUBPROGRAM: A subprogram without reference that supplies specifications and initial values to a main program.</td></tr>
</table>

7.5.3 ACTUAL ARGUMENTS

Actual arguments appear within a CALL statement referencing a SUBROUTINE or in any of the function references. They are associated with the corresponding formal arguments in the auxiliary procedure being referenced and serve to transmit values on a one-to-one basis. Accordingly, formal and actual arguments must agree in order, number, and type, otherwise results are undefined. The permissible forms of actual arguments are the following:

Arithmetic expression

Logical expression

Relational expression

Constant

Simple or subscripted variable

Array name

FUNCTION subprogram name

SUBROUTINE subprogram name

Basic external function name

Intrinsic function name

A calling program statement label is identified by suffixing the label with the character S. This form should be used only when calling DUMP or PDUMP.

> ANSI FORTRAN, X3.9-1966, does not allow a statement label followed by the letter S as a possible actual argument.

Input/output file names may not be used as actual parameters.

The following is allowed:

```
PROGRAM X(OUTPUT, TAPE 6 = OUTPUT)
CALL SUB (6,X)
     .
     .
     .
END
SUBROUTINE SUB(I, B)
     .
     .
     .
WRITE (I, n), B
     .
     .
     .
END
```

7.6 STATEMENT FUNCTION

A statement function is defined by a single expression and applies only to the program or subprogram containing the definition. The name of the statement function is an alphanumeric identifier. A single value is always associated with the name.

A statement function has the form:

name ($p_1, \ldots, p_n$) = E

ANSI FORTRAN, X3.9-1966, does not allow E to contain array references or Hollerith constants.

The p_i are formal arguments and must be simple variables. The p_i cannot be allocated to LEVEL 2 or LEVEL 3. The maximum value of n is 60. E can be any arithmetic or logical expression. It may contain a reference to a library function, statement function, or function subprogram.

During the compilation, the statement function definition is compiled once at the beginning of the program and a transfer is made to this portion of the program whenever a reference is made to the statement function.

A statement function reference has the form:

name ($p_1, \ldots, p_n$)

name is the alphanumeric identifier of the statement function. The actual arguments p_i may be any arithmetic expressions.

The statement function name must not appear in a DIMENSION, EQUIVALENCE, COMMON, or EXTERNAL statement. The name can appear in a type declaration but cannot be dimensioned. Statement function names must not appear as actual or formal arguments or use the same name as the program or subprogram they are in.

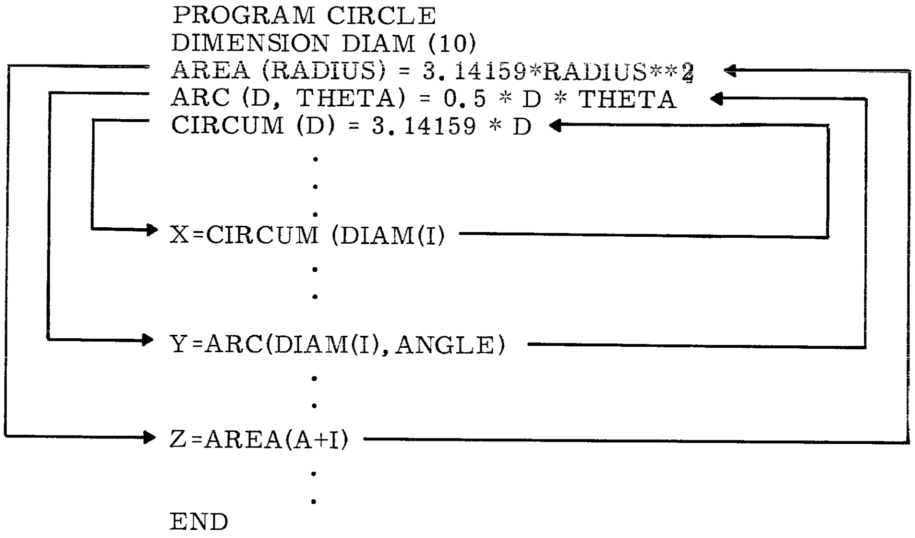
Actual and formal arguments must agree in number, order, and mode. The mode of the evaluated statement function is determined by the name of the arithmetic statement function.

A statement function must precede the first statement in which it is used, but it must follow all declarative statements (DIMENSION, type, etc.) which contain symbolic names referenced in the statement function. All statement functions should precede the first executable statement. Otherwise, an informative diagnostic is provided.

A statement function may not reference itself and if such an attempt is made, a fatal diagnostic is provided.

Example:

The following program calculates various parameters of a set of circles (one to ten). Input is an array of diameters (DIAM). The calculations include the determination of area, arc length, and circumference. These are given by statement functions at the beginning of the program which are referenced as needed.



```
PROGRAM CIRCLE
DIMENSION DIAM (10)
AREA (RADIUS) = 3. 14159*RADIUS**2
ARC (D, THETA) = 0. 5 * D * THETA
CIRCUM (D) = 3. 14159 * D
          .
          .
          .
X=CIRCUM (DIAM(I)
          .
          .
Y=ARC(DIAM(I), ANGLE)
          .
          .
Z=AREA(A+I)
          .
          .
END
```

Explanation: The first reference is contained in the statement

X=CIRCUM(DIAM(I)

in which the subscript I has been determined by calculations in the program. This reference places the actual argument DIAM(I) in the statement function:

CIRCUM(D)=3. 14159*D

via the dummy argument D. The calculation is made and a single value for CIRCUM returned to the referencing statement. The next reference supplies two actual arguments, DIAM (I) and ANGLE, to the statement function for ARC through the dummy arguments D and THETA. A single value for ARC is returned to the referencing statement.

The third reference uses an arithmetic expression, A+I, for an actual argument. This enters the statement function calculation for AREA through the dummy argument RADIUS. A single value for AREA is returned to the referencing statement.

7.7 SUPPLIED FUNCTION

To evaluate frequently used mathematical functions, FORTRAN supplies predefined calculations as well as references to library routines contained in the system. The predefined calculations are called intrinsic or in-line functions and the references to the library routines are called basic external functions.

The intrinsic or in-line function inserts a simple set of calculations into the object program at compile time. The basic external function deals with more complex evaluations by inserting a reference to a library routine in the object program. The names of the supplied functions, their data types, and permissible arguments are predefined (Appendix C). References using these functions must adhere to the format defined in the tables. The type of a supplied function cannot be changed by a type statement.

7.7.1 INTRINSIC FUNCTIONS

An intrinsic function is a compiler-defined set of calculations that is inserted in the referencing program at compile time. The form of the intrinsic function and its reference are identical to the statement function outlined above. The table in Appendix C lists the intrinsic functions available.

The name of an intrinsic function listed in this table must satisfy all of the following requirements:

The name must not appear in an EXTERNAL statement or be the name of a statement function

The name must not appear in a type statement declaring it to be other than the type specified in the table

Every appearance of the name must be followed by a list of parameters of correct type enclosed in parentheses, unless the name is in a type statement

7.7.2 BASIC EXTERNAL FUNCTIONS

A basic external function is a call on one of the predefined library routines included with the system. These library routines are used to evaluate standard mathematical functions such as sine, cosine, square root, etc. A basic external function is referenced by the appearance of the function name with appropriate arguments of correct type in an arithmetic or logical statement. A list of basic external functions is given in Appendix C.

7.8 SUBPROGRAMS

Subprograms are used to implement programming capability beyond the limitations of supplied functions and the statement function. Although written as a subset of another program, the subprogram is compiled separately. It has its own independent variables, and its use is not limited to communication with the program for which it was written. Procedure subprograms handle routine calculations unique to the user. Specification subprograms are used to enter values into COMMON and supply program specifications.

Procedure subprograms are of two kinds: FUNCTION and SUBROUTINE. The FUNCTION subprogram is referenced by the appearance of its name in the calling program. The SUBROUTINE subprogram is referenced by a CALL statement in the calling program. A procedure subprogram returns control to a calling program through one or more RETURN statements. Because they are independent programs, procedure subprograms must terminate with an END statement to signal to the compiler that the physical end of the source program has been reached. An END statement is generated as a STOP. If a procedure subprogram does not contain at least one RETURN statement, the successful execution of that subprogram will terminate the entire program.

The fundamental difference between FUNCTION and SUBROUTINE subprograms is given in Table 7-2.

There is one type of specification subprogram, the BLOCK DATA subprogram.

TABLE 7-2. DIFFERENCES BETWEEN A FUNCTION AND SUBROUTINE SUBPROGRAM

Function	Subroutine
Referenced by the name appearing in an arithmetic or logical statement and returns a value to be used as an operand at the point of reference	Referenced by a CALL statement
Must have one or more arguments	Need not have any arguments
Name is typed by first letter or by the type designation appearing before the word FUNCTION	No type associated with name

7.8.1 FUNCTION SUBPROGRAM

A FUNCTION subprogram is a collection of FORTRAN statements headed by a FUNCTION statement and written as a separate program to perform a set of calculations when its name appears in the referencing program. The mode of the function is determined by a type indicator or the name of the function. The first statement of a FUNCTION subprogram must be one of the following forms where name is an alphanumeric identifier and the p_i are formal arguments with n assuming any integer value up to 60. A FUNCTION statement must have at least one argument.

Example:

FUNCTION name $(p_1, \ldots, p_n)$

type FUNCTION name $(p_1, \ldots, p_n)$

FORTRAN II FUNCTION name $(p_1, \ldots, p_n)$

FORTRAN II type FUNCTION name $(p_1, \ldots, p_n)$

ANSI FORTRAN, X3.9-1966, does not specify FORTRAN II.

Type is REAL, INTEGER, DOUBLE PRECISION, DOUBLE, COMPLEX, or LOGICAL. When the type indicator is omitted, the type is determined by the first character of the function name.

> ANSI FORTRAN, X3.9-1966, does not specify DOUBLE as a replacement for DOUBLE PRECISION.

The FUNCTION name must not appear in a DIMENSION statement or an array declaration. FUNCTION must be assigned a value by appearing at least once in the subprogram as one of the following:

Left-hand identifier of a replacement statement

A DO index variable

An element of an input list

An actual argument of a subroutine reference

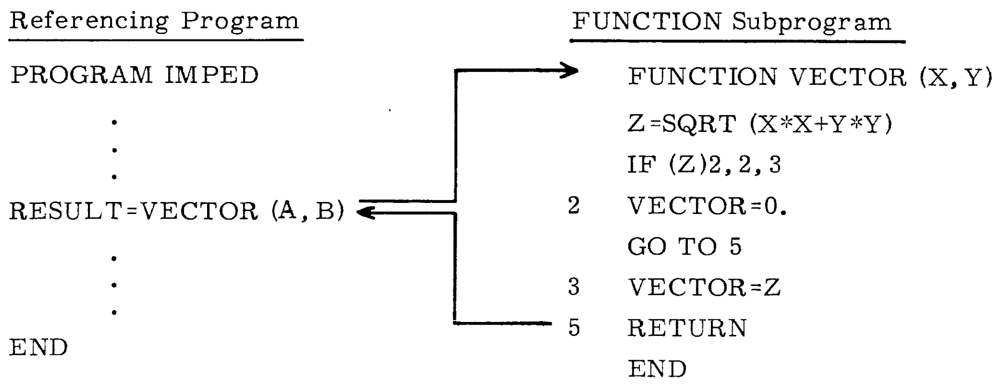
If not, the value returned is undefined. The name of a FUNCTION must not appear in an array declaration.

The FUNCTION subprogram accepts arguments from the referencing program through the argument list and returns a value through the FUNCTION name. The FUNCTION subprogram may define and redefine one or more arguments and return these values as is done in a SUBROUTINE (paragraph 7.8.2).

> ANSI FORTRAN, X3.9-1966, does not allow an assign variable to be an actual argument or to be in COMMON.

When a FUNCTION reference is encountered in an expression, control transfers to the FUNCTION subprogram indicated. When RETURN is encountered in the FUNCTION subprogram, control returns to the statement containing the FUNCTION reference. The value of the FUNCTION is the value of function name.

Example:



```
Referencing Program                    FUNCTION Subprogram

PROGRAM IMPED                               FUNCTION VECTOR (X,Y)
      .                                     Z=SQRT (X*X+Y*Y)
      .                                     IF (Z)2,2,3
      .                                 2   VECTOR=0.
RESULT=VECTOR (A,B)                         GO TO 5
      .                                 3   VECTOR=Z
      .                                 5   RETURN
      .                                     END
END
```

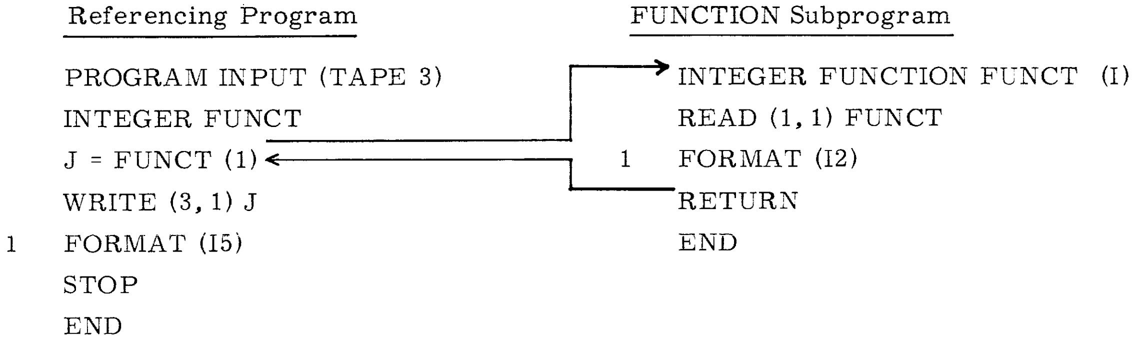
The FUNCTION subprogram is referenced by the appearance of the name and list in the statement

```
RESULT=VECTOR (A,B)
```

The values represented by the actual arguments A and B are communicated to the subprogram through the dummy arguments X and Y.

The first calculation in the subprogram involves the appearance of a secondary reference: SQRT. This reference passes the calculated value in the parentheses to the basic external function for obtaining a square root. The result is returned to the subprogram and placed in storage location Z. Z is then tested to see if it is positive. If not, the function name VECTOR is equated to zero and that value is returned to the reference; if it is positive, the function name VECTOR is equated to that positive value and returned to the reference.

The following example shows how a FUNCTION subprogram can establish a value for the FUNCTION name by using an input statement rather than an arithmetic statement.



```
     Referencing Program                      FUNCTION Subprogram

     PROGRAM INPUT (TAPE 3)                   INTEGER FUNCTION FUNCT  (I)
     INTEGER FUNCT                            READ (1,1) FUNCT
     J = FUNCT (1)                        1   FORMAT (I2)
     WRITE (3,1) J                            RETURN
1    FORMAT (I5)                              END
     STOP
     END
```

Since the subprogram is intended to deal with integer values and its name is implicitly real, the name is typed integer in the referencing program and in the FUNCTION statement of the subprogram. The subprogram is referenced by the statement:

```
J = FUNCT (1)
```

which arbitrarily passes the constant 1 as an actual argument. It enters the subprogram through the dummy argument I in the FUNCTION statement but is never used. This step is performed solely to satisfy the requirements of a FUNCTION subprogram. The subprogram reads in the value from a card and stores it in the location designated by the name of the FUNCTION subprogram. There it is available to the referencing program which stores it in J and then prints it out.

7.8.2 SUBROUTINE SUBPROGRAM

A SUBROUTINE subprogram is a collection of FORTRAN statements headed by a SUBROUTINE statement and written as a separate program to perform a set of calculations when called by a referencing program. It may return none, one, or more values. A value or type is not associated with the subroutine name itself.

The first statement of a subroutine subprogram must have one of the following forms:

FORTRAN II SUBROUTINE name($p_1, \ldots, p_n$)

SUBROUTINE name($p_1, \ldots, p_n$)

name	alphanumeric identifier and p_i are formal arguments; n may be 1 to 60
p_i	parameter list, optional

The SUBROUTINE subprogram may accept arguments from the calling program and can return results through its arguments or in COMMON.

Example:

```
Referencing Program                 Subprogram

PROGRAM TENSOR                   ┌──> SUBROUTINE MATRIX
COMMON/BLK1/X(20,20),            │    COMMON/BLK1/A(20,20),
  Y(20,20),Z(20,20)              │      B(20,20),C(20,20)
        .                        │          .
        .                        │          .
        .                        │          .
CALL MATRIX ─────────────────────┘    DO 10 I = 1,20
Next statement <─────────────────┐    DO 10 J = 1,20
        .                        │    C (I,J) = 0.0
        .                        │    DO 10 K = 1,20
        .                        │ 10 C(I,J) = C(I,J) + A(I,K) *B(K,J)
END                              └────RETURN
                                      END
```

The referencing program reserves storage for three successive arrays in the labeled COMMON block. It is assumed that two of these arrays, X and Y, have values stored in them before the CALL statement is reached. The CALL statement transfers control to the subroutine without passing any arguments. The subroutine performs the matrix multiplication of the first two arrays and stores the results in the third. Control is returned to the next statement after the CALL in the referencing program. The subroutine obtains the values for its calculations from the labeled common block and returns the results it derives to the same labeled common block.

CALL SSWTCH (i, j)

If sense switch i is down, set j = 1. If sense switch i is up, set j = 2. i is 1 to 6. If i is out of the range, the results are undefined.

CALL OVERFL (j)[†]

If a floating point overflow condition exists, set j = 1. If no overflow exists, set j = 2; set the machine to a no overflow condition.

CALL DVCHK (j)[†]

If division by zero occurred, set j = 1 and clear the indicator; if division by zero did not occur, set j = 2.

CALL SECOND (t)

Returns CP time from start of job in seconds in floating point format to three decimal places. t is a real variable.

CALL EXIT

Terminate program execution and return control to the monitor.

Format:

CALL DUMP ($a_1, b_1, f_1, \ldots, a_n, b_n, f_n$)

CALL PDUMP ($a_1, b_1, f_1, \ldots, a_n, b_n, f_n$) $(n \leq 20)$

Dump storage on OUTPUT file in indicated format. For PDUMP, control returns to the calling program; for DUMP, execution terminates and control returns to the operating system. If no arguments are provided, an octal dump of all storage occurs.

The a_i and b_i are SCM core addresses, variables, or statement numbers. They indicate the first word and the last word of the storage area to be dumped.

The statement numbers must be 1 to 5 digits trailed by an S; CALL DUMP (10S, 20S, 0). If b_i is the last statement of a DO loop, then b_iS is not allowed to be used as the last word of the storage area to be dumped.

ANSI FORTRAN, X3.9-1966, does not allow a statement label followedby the letter S as a possible argument.

[†] Currently J is always set to 2 (see LEGVAR in Appendix C).

The dump format indicators are as follows:

f = 0 or 3 octal dump

f = 1 real dump

f = 2 integer dump; if bit 48 is set (normalize bit)

CALL READEC (cm, lcm, n)

Transfers words from LCM into SCM.

cm SCM address; array or variable name

lcm LCM relative address; array or variable name

n Count of the number of words to be transferred; must be an integer variable or integer constant

CALL WRITEC (cm, lcm, n)

Transfers word from SCM to LCM as for READEC.

cm SCM address; array or variable name

lcm LCM relative address; array or variable name

n Count of the number of words to be transferred; must be an integer variable or integer constant

Example:

```
PROGRAM ECS (INPUT,OUTPUT)
DIMENSION A (1000)
DIMENSION B (1000)
LEVEL 3, B
COMMON/LCM/B
CALL WRITEC(A, B, 1000)
C   TRANSFER 1000 CM WORDS BEGINNING AT SCM LOCATION
C   A INTO LCM BEGINNING AT LOCATION B OF THE USERS
C   LCM AREA.
END
```

```
CALL OPENMS (u, ix, ℓ, p)
CALL READMS (u, fwa, n, i)
CALL WRITMS (u, fwa, n, i)
CALL STINDX (u, ix, ℓ )
CALL CLOSMS (u)
```

Control transmission between SCM and a mass storage device u Logical unit number

ix First word address of the index (in SCM)

ℓ Length of index

$\ell \geq 2$ (number of index entries)+1 for name index, $\ell \geq$ number of index entries + 1 for number index

p=1 Indicates file is referenced through a name index, p=0 indicates a number index

fwa SCM address of the first word of the record

n Number of SCM words to be transferred

i Record number or address. When address, it is the address of record number or record name. Record number is right justified and record name is left justified display code 1-7 characters.

OPENMS is used to open the mass storage file. This routine informs SCOPE that this file is a random access file; and, if the file exists, the master index is read into the area specified by the program.

READMS and WRITMS perform the data transfers to and from SCM.

STINDX is called to change the file index to the base specified in the CALL.

CLOSMS is used to close the mass storage file and write out the index and the control word to the file. The control word is the first word of the file. It points to the index which is the last record. The operations performed by CLOSMS will be automatically performed by STOP or END.

7.9 CALL STATEMENT

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

CALL name
or
CALL name ($p_1,\ldots,p_n$)

name is the name of the subroutine being called, and p is an actual argument; n is 1 to 60. The name should not appear in any declarative statement in the calling program except the EXTERNAL statement when name is also an actual argument.

The CALL statement transfers control to the subroutine. When a RETURN statement is encountered in the subroutine, control is returned to the next executable statement following the CALL statement in the calling program. If the CALL statement is the last statement in a DO loop, looping continues until the DO loop is satisfied. The CALL statement is executed each time the terminal statement is reached.

Examples:

1.

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

Calls

```
        CALL BLDX(X(I),Y(I,W)
        CALL BLDX(X(I)+H/2.,Y(I) + C(J),PROX)
        CALL BLDX(SIN(Q5,EVEC(I+J),OVEC(L)
```

2.

```
        SUBROUTINE MATMULT
        COMMON/ITRARE/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
```

Operations in MATMULT are performed on variables contained in the common block ITRARE. This block must be defined in all calling programs.

```
        COMMON/ITRARE/AB(20,20),CD(20,20),EF(20,20)
        CALL MATMULT
```

3.

```
        SUBROUTINE AGMT (SUB,ARG)
        COMMON/ABL/XP(100)
        ARG=0
        DO 5 I=1,100
   5    ARG=ARG +XP(I)
        CALL SUB
        RETURN
        END
```

Here the dummy argument SUB is used to transmit another subprogram name. The call to SUBROUTINE AGMT might be CALL AGMT (MULT,FACTOR), where MULT is specified in an EXTERNAL statement (paragraph 7.10).

7.10 EXTERNAL STATEMENT

When the actual argument list which calls a function or subroutine program contains a function or subroutine name, that name must be declared in an EXTERNAL statement.

EXTERNAL $name_1, name_2, \ldots$

The EXTERNAL statement must precede the first statement of any program which calls a function or subroutine subprogram using the EXTERNAL name. When it is used, EXTERNAL always appears in the calling program; it may not be used with statement functions. If it is, a compiler diagnostic is provided.

Examples:

1. A function name used as an actual argument requires an EXTERNAL statement.

<u>Calling Program Reference</u>

```
          .
          .
          .
EXTERNAL SIN
CALL PULL(SIN,R,Q)
          .
          .
          .
```

<u>Called Subprogram</u>

```
SUBROUTINE PULL(X,Y,Z)
          .
          .
Z=X(Y)    .
          .
          .
          .
```

But a function reference used as an actual argument does not need an EXTERNAL statement.

<u>Calling Program Reference</u>

```
          .
          .
          .
CALL PULL(SIN(R),Q)
          .
          .
          .
```

<u>Called Subprogram</u>

```
SUBROUTINE PULL(X,Z)
          .
          .
          .
Z=X
          .
          .
          .
END
```

2. A subroutine used as an actual argument must have its name declared in an EXTERNAL statement in the calling program.

```
COMMON/ABL/ALST(100)
EXTERNAL RTENTA, RTENTB
CALL AGMT(RTENTA,V1)
CALL AGMT(RTENTB,V1)
```

When a subprogram name appears as an actual argument, any arguments to be associated with a call of this subprogram can be passed via actual arguments.

Examples:

Calling Program

```
EXTERNAL ADDER
        .
        .
        .
CALL SUB(ADDER,A,B)
        .
        .
        .
```

Called Subprogram

```
SUBROUTINE SUB(X,Y,Z)
        .
        .
        .
CALL X(Y,Z)
        .
        .
        .
END
```

CALL SUB(ADDER(A,B)) would imply that ADDER is a function reference, not a subroutine name.

7.11 ENTRY STATEMENT

The statement provides alternate entry points to a FUNCTION or SUBROUTINE subprogram.

```
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 dummy arguments, 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 or as the dependent statement of a logical IF; the ENTRY statement cannot be labeled. The first executable statement following ENTRY becomes an alternate entry point to the subprogram.

In the calling program, the reference to the entry name is made just as though reference were being made to the FUNCTION or SUBROUTINE in which the ENTRY is embedded. The name may appear in an EXTERNAL statement, and if it is a function entry name, in a type statement.

The ENTRY name may not be given type explicitly in the defining program; it assumes the same type as the name in the FUNCTION statement.

Examples:

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

This could be called from the main program as follows:

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

ANSI FORTRAN, X3.9-1966, does not specify the ENTRY statement.

7.12 VARIABLE DIMENSIONS IN SUBPROGRAMS

In many subprograms, especially those performing matrix manipulation, the programmer may wish to vary array dimensions each time the subprogram is called.

This is accomplished by specifying the array name and its dimensions as dummy arguments in the FUNCTION or SUBROUTINE statement. The corresponding actual arguments specified in the calling program are used by the called subprogram. The maximum dimensions that any given array may assume are determined by dimensions in a DIMENSION, COMMON, or type statement in the program.

The dummy arguments representing the array dimensions must be simple integer variables. The array name must also be a dummy argument. The actual argument representing the array dimensions must have integer values.

ANSI FORTRAN, X3.9-1966, does not allow the array dimension to be changed during execution of the subprogram.

The total number of elements of the corresponding array in the subprogram may not exceed the total number of elements of a given array in the calling program.

Example:

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)
      END
```

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

```
      DIMENSION A(10,10),B(10,10),C(10,10),E(5,5),F(5,5),G(5,5),H(10,10)
```

the program may call the subroutine MATADD from several places within the main program as follows:

```
      CALL MATADD(A,B,C,10,10)
      CALL MATADD(E,F,G,5,5)
      CALL MATADD(B,C,A,10,10)
      CALL MATADD(B,C,H,10,10)
```

The compiler does not check to see whether the limits of the array established by the DIMENSION statement in the main program are exceeded.

7.13 PROGRAM ARRANGEMENT

FORTRAN assumes that all statements and comments appearing between a PROGRAM, SUBROUTINE, or FUNCTION statement and an END statement belong to one program (see Appendix E).

Format:

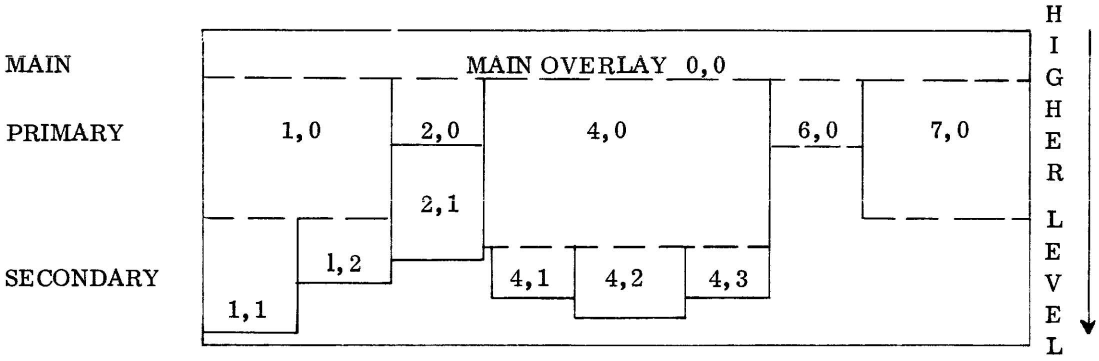
```
    PROGRAM WHAT
          .
          .
          .
    END
    SUBROUTINE S1(A, B)
          .
          .
          .
    END
    SUBROUTINE S2
          .
          .
          .
    END
    REAL FUNCTION F1(P1)
          .
          .
          .
    END
```

OVERLAYS 8

An overlay is a portion of a program written on a file in absolute form and loaded at execution time without delay for relocation. The user defines an overlay with the OVERLAY card. He calls it with the CALL OVERLAY statement.

8.1 LEVELS

Levels are used to describe the sequence of loading overlays and to specify which sections of code are to overlay others. In SCOPE 2, there are three levels of overlaying, main, primary, and secondary. Up to three overlays may be in core simultaneously. They are usually loaded contiguously. The primary or secondary levels may be replaced by other overlays. The following diagram demonstrates the relationship of the levels when they are loaded into core. This example shows a number of different core loads which might exist for a single job.



8.2 IDENTIFICATION

Overlays may be loaded from specified files. A single overlay may be loaded only from a single file, although many files may be used for loading by a single job. An overlay is identified by its level number. The level number is a pair of two-digit octal numbers (0-77). The first number is the primary level, the second is the secondary level. An overlay with a non-zero primary level and a zero secondary level (1, 0) is a primary overlay. Any overlay with the same primary level and a non-zero secondary level (1, 1) is associated with and subordinate to the corresponding primary and is called a secondary overlay. This difference is significant when overlays are loaded. Level (0, 0) is reserved for the initial or main overlay which is neither primary nor secondary; it is a special case which remains in memory during overlay execution. Overlay numbers (0, 1) to (0, 77) are illegal, that is, the main overlay cannot have any secondaries.

The main overlay (0, 0) is loaded first. All primary overlays, when called, are loaded at the same point immediately following the main overlay. Secondary overlays are loaded immediately following their associated primary overlays. Loading the next primary overlay destroys the first loaded primary overlay and any associated secondary overlays. Likewise, the loading of a secondary overlay destroys a previously loaded secondary overlay.

8.3 COMPOSITION

Each overlay must have at least one program having the characteristics of a FORTRAN main program.

An overlay may consist of one or more FORTRAN or COMPASS programs. The program name becomes the primary entry point for the overlay through which control passes when the overlay is called. An overlay cannot reference entry points in higher level overlays; for example, (1, 0) the correct reference entry points are (1, 5). The only method of reference for a main overlay to primary and secondary overlays is through the CALL OVERLAY statement. However, the primary overlay may reference any entry point in the main overlay, while the secondary overlay may reference any entry point in the primary or main overlay.

Blank common and labeled common may be defined in any level overlay and referenced by that overlay and higher level overlays. (The same rules apply as for entry points.)

Blank common is allocated at the top (highest address) of the first overlay in a linear tree in which blank common is declared. That is, if blank common is declared in the (0, 0) overlay, it will be allocated at the top of the (0, 0) overlay and will be accessible to all overlays in that overlay structure. If blank common is not declared in the (0, 0) overlay and is declared in the (1, 0) overlay, it will be allocated at the top of the (1, 0) overlay and will be accessible only to the associated (1, X) overlays.

Labeled common blocks are generated in the overlay in which they are first encountered. They may be referenced by all overlays which are higher in the same linear structure. They may only be preset in the overlay in which they are generated.

LCM common blocks must be defined and preset in the main overlay. The entire overlay structure may reference LCM COMMON block.

An overlay is established by an OVERLAY card which precedes the program cards. The overlay consists of all programs appearing between the OVERLAY card and the next OVERLAY card or an end of file or an end of record.

8.4 CALL

Overlays are called by the following statement.

CALL OVERLAY (fn, ℓ_1, ℓ_2, p)

OVERLAY is a FORTRAN subroutine which translates the FORTRAN call into a call to the loader

fn — Variable name of the location containing the name of the file (left justified display code) which includes the overlay

ℓ_1 — Primary level of the overlay

ℓ_2 — Secondary level of the overlay

p — Recall parameter. If p equals 6HRECALL, the overlay is not reloaded when it is in memory

The first three parameters must be specified; the absence of any one could result in a mode error at execution time. The levels appearing on the OVERLAY loader card are always octal. The normal mode for parameters in FORTRAN calls is decimal. This fact should be considered when coding the ℓ_1, ℓ_2 parameters. The programmer can keep his level numbers straight by using octal notation on both control and call cards.[†]

If uniqueness is ensured at execution time, more than one overlay may be created with the same level numbers. Uniqueness is determined by the level numbers, the file name from which the overlay is to be loaded, and the position of the overlay on the file. Since the loader selects the first overlay encountered on the specified file with level numbers which match those in the call, it is possible to position a number of overlays on a file with the same identifier and by properly sequencing the calls thereto, have available a number of different overlays.

Loading from a file may require and end-around search of the file for the specified overlay; this can be time consuming in large files. When speed is essential, each overlay should be written to a separate file.

8.5 LOADER CARDS

Loader cards are processed directly by the loader. They provide the loader with information necessary for generating overlays. All loader cards must precede the subprogram text to be loaded. Formats are the same as for SCOPE control cards. However, if they are in the FORTRAN decks, the loader cards must be punched in columns 7 through 72.

[†] Level numbers given in the CALL OVERLAY, however, are decimal; for example, the overlay card for overlay 1, 9 would be OVERLAY(fn, 1, 11) and its call would be CALL OVERLAY(fn, 1, 9).

8.6 OVERLAY CARDS

OVERLAY (fn, ℓ_1, ℓ_2, Cnnnnnn)

fn — File name onto which the generated overlay is to be written

ℓ_1 — Primary level number

ℓ_2 — Secondary level number

(ℓ_1, ℓ_2) must be (0, 0) for first overlay card and must be in octal[†]

Cnnnnnn — optional; nnnnnn can be up to 6 octal digits. If absent, overlay is loaded normally. If present, overlay is loaded nnnnnn words from the start of blank common. This provides a method for changing the size of blank common at execution time. Cnnnnnn cannot be an OVERLAY 0, 0 card on a primary directive if the main overlay has no blank common, and on a secondary directive if the associated primary has no blank common.

The first overlay card must have an fn. Subsequent cards may omit fn, and the overlay is written on the same fn.

Each overlay card must be followed by a program card. The program card for the main overlay must specify all needed file names, such as INPUT, OUTPUT, and TAPE 1, for all overlay levels. File names should not appear in program cards for other than the (0, 0) OVERLAY.

The groups of relocatable decks processed by the loader in forming overlays must be presented to the loader in proper order. This requires that the 0, 0 overlay group be first. The next order may be any primary group followed by all of its associated secondary groups, then any other primary group followed by its associated secondary groups, et cetera.

8.7 RETURN FROM OVERLAY

Control is returned from a primary overlay to the main overlay or from a secondary overlay to primary overlay by using a RETURN statement in the main program of the overlay. Control is returned to the statement which follows CALL OVERLAY.

[†] Level numbers given in the CALL OVERLAY, however, are decimal; for example, the overlay card for overlay 1, 9 would be OVERLAY(fn, 1, 11) and its call would be CALL OVERLAY(fn, 1, 9).

Example:

```
EXAM, S70.
RUN(S)
LGO.
7/8/9
OVERLAY(XFILE, 0, 0)
PROGRAM ONE(INPUT, OUTPUT, PUNCH)
     .
     .
     .
CALL OVERLAY(5HXFILE, 1, 0)
     .
     .
     .
STOP
END
OVERLAY(XFILE, 1, 0)
PROGRAM ONE ZERO
CALL OVERLAY(5HXFILE, 1, 1)
     .
     .
     .
RETURN
END
OVERLAY(XFILE, 1, 1)
PROGRAM ONE ONE
     .
     .
     .
RETURN
END
6/7/8/9
```

INPUT/OUTPUT FORMATS 9

Data is transferred between storage and the files for external units in one of two modes, formatted and unformatted. Formatted transmissions are dependent on the structure of the data they contain and as such must have their format specified. This is accomplished by means of a FORMAT statement. Unformatted data is transferred as a single string and does not require a format specification. Both forms require an I/O statement that identifies the unit involved and specifies the list of data to be moved.

9.1 INPUT/OUTPUT LIST

The list portion of an I/O statement indicates the data items and the order, from left to right, of transmission. The I/O list can contain any number of elements. List items may be array names, simple or subscripted variables, or an implied DO loop. Items are separated by commas, and their order must correspond to any FORMAT specification associated with the list. External records are always read or written until the list is satisfied.

Subscripts in an I/O list may be any of the following forms in which c and k are unsigned integer constants and v is a simple integer variable:

Form	Example
(c)	(4)
(v)	(I)
(v±k)	(J+3)
(c*v)	(5*K)
(c*v±k)	(2*L-8)

Examples:

```
READ 100,A,B,C,D
READ 200,A,B,C(I),D(3,4),E(I,J,7),H
READ 101,J,A(J),I,B(I,J)
READ 102,DELTA(5*J+2,5*I-3,5*K),C,D(I+7)
```

The integer variable in a list must be previously defined, or it must be defined within an implied DO loop in the list.

Examples:

```
    READ 300,A,B,C,(D(I),I=1,10),E(5,7)F(J),(G(I),H(I),I=2,6,2)
    READ 400,I,J,K(((A(II,JJ,KK),II=1,I),JJ=1,J),KK=1,K)
    READ 500,((A(I,J),I=1,10,2),B(J,1),J=1,5),E,F,G(L+5,M-7)
```

9.2 ARRAY TRANSMISSION AND IMPLIED DO LOOPS

An entire array or any part of an array can be transferred as a single specification in an I/O list by using an implied DO loop. In general, an implied DO loop is a list followed by a comma.

Format:

$(((A(I, J, K), L_1=m_1, m_2, m_3), L_2=n_1, n_2, n_3), L_3=p_1, p_2, p_3)$

m_i, n_i, p_i	Unsigned integer constants or simple integer variables. If m_3, n_3, or p_3 is omitted, it is assumed equal to 1.
I, J, K	Subscripts of A
L_1, L_2, L_3	Index variables I, J, K in same order

A DO implied specification is of one of the forms:

$i = m_1, m_2, m_3$

$i = m_1, m_2$

where i, m_1, m_2, m_3, are defined as for a DO statement. The range of the DO implied specification is the list of the implied DO loops. The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in an implied DO loop are specified for each cycle of the implied DO.

An array name which appears without subscripts in an I/O list causes transmission of the entire array by columns.

Example:

```
    DIMENSION B(10,15)
    the statement
        READ 13, B
    is equivalent to
        READ 13,((B(I,J),I=1,10),J=1,15)
```

An implied DO loop can be used to transmit a simple variable more than one time. For example, the list item (A(K), B, K=1, 5) causes the transmission of the variable B five times. A list of the form K, (A(I), I=1, K) is permitted and the input value of K is used in the implied DO loop. The index variable in an implied DO list in an I/O DATA statement should be an implicit integer.

Examples:

1. Simple implied DO loop list items.

```
          READ 400,(A(I),I=1,10)
    400   FORMAT(E20.10)
```

   This statement is equivalent to the following DO loop.

```
          DO 5 I=1,10
    5     READ 400, A(I)
          READ 100,((A(JV,JX),JV=2,20,2),JX=1,30)
          READ 200,(BETA(3*JON+7),JON=JONA,JONB,JONC)
          READ 300,(((ITMSLST(I,J+1,K-2),I=1,25),J=2,N),K=IVAR,IVMAX,4)
          READ 600,(A(I),B(I),I=1,10)
    600   FORMAT(F10.2,E6.1)
```

   The previous statement is equivalent to the DO loop:

```
          DO 17 I=1,10
    17    READ 600,A(I),B(I)
```

2. Nested implied DO list items.

```
        READ 100,((((A(I,J,K),B(I,L),C(J,N),I=1,10),J=1,5),K=1,8),L=1,15),N=2,7)
```

   Data is transmitted in the following sequence:

```
        A(1,1,1),B(1,1),C(1,2),A(2,1,1),B(2,1),C(1,2)...
        ...A(10,1,1),B(10,1),C(1,2),A(1,2,1),B(1,1),C(2,2)...
        ...A(10,2,1),B(10,1),C(2,2)...A(10,5,1),B(10,1),C(5,2)...
        ...A(10,5,8),B(10,1),C(5,2)...A(10,5,8),B(10,15),C(5,2)...
        ...A(10,5,8),B(10,15),C(5,7)
```

   The following list item will transmit the array E(3,3) by columns:

```
        READ 100,((E(I,J),I=1,3),J=1,3)
```

   The following list item will transmit the array E(3,3) by rows:

```
        READ 100,((E(I,J),J=1,3),I=1,3)
```

```
3.          DIMENSION MATRIX (3,4,7)
            READ 100, MATRIX
      100   FORMAT (I6)
```

The above items are equivalent to the following statements:

```
            DIMENSION MATRIX(3,4,7)
            READ 100,(((MATRIX(I,J,K),I=1,3),J=1,4),K=1,7)
```

The list is equivalent to the nest of DO loops:

```
            DO 5 K =1,7
            DO 5 J =1,4
            DO 5 I =1,3
      5     READ 100, MATRIX(I,J,K)
```

9.3 FORMAT DECLARATION

Formatted I/O statements required a FORMAT declaration which contains conversion and editing information relating to internal/external structure of the corresponding I/O list items. A FORMAT declaration has the following form:

FORMAT (spec$_1$, . . . , k(spec$_m$, . . .), spec$_n$, . . .)

spec$_i$	Format specification
k	Optional repetition factor; must be unsigned integer constant

The FORMAT declaration is nonexecutable and may appear anywhere in the program. FORMAT declarations must have a statement label in columns 1-5.

The data items in an I/O list are converted from one representation to another (external/internal) according to FORMAT conversion specifications. FORMAT specifications may also contain editing codes.

Conversion specifications:

srEw.d	Single precision floating point with exponent
srEw.dEe	With explicitly specified exponent length
srEw.dDe	With explicitly specified exponent length
srFw.d	Single precision floating point without exponent
srGw.d	Single precision floating point with or without exponent
srDw.d	Double precision floating point with exponent
rIw	Decimal integer conversion
rIw.z	With minimum number of digits specified
rLw	Logical conversion
rAw	Character conversion
rRw	Character conversion
rOw	Octal integer conversion
rOw.d	With minimum number of digits specified
rZw	Hexadecimal conversion
srVw.d	Variable type conversion

E, F, G, D, I, L, A, R, O, and Z are the codes which indicate the type of conversion.

w	Non-zero, unsigned, integer constant which specifies the field width in number of character positions in the external record. This width includes any leading blanks, +, or - signs, decimal point, and exponent.
d	Integer constant which represents the number of digits to the right of the decimal point within the field. On output all numbers are rounded.
r	Unsigned integer constant which indicates the conversion code is to be repeated.
s	Optional; it represents a scale factor.
z	Minimum number of digits to output.

The field width w must be specified for all conversion codes. If d is not specified for w.d, it is assumed to be zero. w must be $\geq$ d.

Complex data items are converted on I/O according to a pair of consecutive Ew.d or Fw.d specifications.

Editing specification

wX†	Intraline spacing (refer to 9.6.1)	/	Begin new record (refer to 9.6.4)
$wHh_1,h_2,\ldots,h_n$	Heading and labeling (refer to 9.6.2 and 9.6.3)	*...*	Heading and labeling (refer to 9.6.6)
Tn	Column selections (refer to 9.6.7)	'...'	Heading and labeling (refer to 9.6.5)

The variable n is an unsigned integer ranging from 0 to 136.

> ANSI FORTRAN, X3.9-1966, requires that the field width w must always be specified. In the w.d form, d must be specified also. Further w must be greater than or equal to d. The editing specifications wX and wHh must be separated by a slash or comma. The 3 parameter and the formats Ew.dEe, Ew,dDe, Vw.d, and Zw are not recognized by ANSI.

Examples:

```
          COMPLEX A,B
          PRINT 10,A
   10     FORMAT(F7.2,F9.2)
          READ 11,B
   11     FORMAT(E10.3,E10.3)
```

9.4 CONVERSION SPECIFICATION

9.4.1 Ew.d, Ew.dEe, AND Ew.dDe OUTPUT

Real numbers in storage are converted to the character form for output with the E conversion. The field occupies w positions in the output record with the real number right justified in the form:

s.a...a$\pm$eee $|eee| \geq 100$

or

s.a...aE$\pm$ee $0 \leq ee \leq 99$

s indicates no character position or minus. a's are the most significant digits of the value of the data output 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.

†Only w may be negative for the X specification.

Field w must be wide enough to contain the significant digits, sign, decimal point, E, and the exponent. Positive numbers need not reserve a space for the sign of the number. Generally, w≥d+6 or w≥d+e+4 for negative numbers and w≥d+5 or w≥d+e+3 for positive numbers.

If the field is not wide enough to contain the output value, asterisks are inserted in the whole field. If the field is longer than the output value, the quantity is right justified with blank fill to the left. Double precision numbers cannot be output using Ew.d.

> ANSI FORTRAN, X3. 9-1966, does not enter asterisks if the field is too narrow.

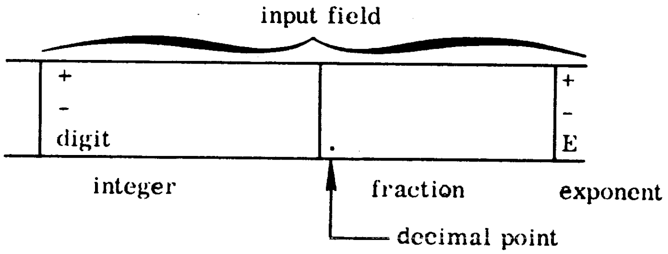
Examples:

```
              PRINT 10,A                        A contains -67.32
    10        FORMAT(1X,E9.3)                        or     +67.32
                   Result:  -.673E+02    or    b.673E+02
              PRINT 10,A
    10        FORMAT (1X,E11.4)
                   Result:  b-.6732E+02  or    bb.6732E+02
              PRINT 10,A                        A contains -67.32
    10        FORMAT (1X,E8.3)                  Provision not made for sign
                   Result:  ********
              PRINT 10,A                        A contains -67.32
    10        FORMAT (1X,E9.4)
                   Result:  *********           Increasing significance requires more
                                                total width
              PRINT 10,A                        A contains -67.32
    10        FORMAT (1X,E10.4)
                   Result:  -.6732E+02
```

9.4.2 Ew.d, Ew.dEe, AND Ew.dDe INPUT

The E specification converts the number in the input field to a real number and stores it in the proper location.

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; blanks are interpreted as zeros. An all blank field is interpreted as minus zero. The range of permissible values is |3.13152 |E-294 to |1.26501 | E322 approximately. Smaller numbers will be treated as zero; larger numbers will cause a fatal error message.

The integer subfield begins with a sign (+ or -) or a digit and may contain a string of digits. The integer field is terminated by a decimal point, D, E, +, -, or the end of the input field.

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

The exponent subfield may begin with D, E, + or -. When it begins with D or E, the + is optional between D or E and the string of digits of the subfield. The value of the string of digits in the exponent subfield must be less than 323.

Format:

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

In the Ew.d specification, d acts as a negative power-of-ten scaling factor when an external decimal point is not present. The internal representation of the input quantity is:

$$(\text{integer subfield})\text{x}10^{-d}\text{x}10^{(\text{exponent subfield})}$$

For example, if the specification is E7.8, the input quantity 3267+05 is converted and stored as $3267\text{x}10^{-8}\text{x}10^{5} = 3.267$.

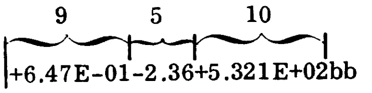
A decimal point in the input field overrides d. The input quantity 3.67294+5 read by an E9.d specification is always stored as $3.6729\text{x}10^{5}$. When d does not appear, it is assumed to be zero. If e is specified, it is ignored.

The field length specified by w in Ew.d should always be the same as the length of the field containing the input number. When it is not, incorrect numbers may be read, converted, and stored as shown below without any indication of error given to the user. The field w includes the significant digits, signs, decimal point, E or D, and exponent.

Example:

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

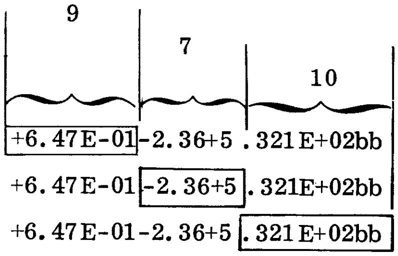
Input quantities on the card are in three contiguous fields columns 1 through 24:



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

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

Reading proceeds as follows:


9
7
10
+6.47E-01-2.36+5 .321E+02bb
+6.47E-01-2.36+5 .321E+02bb
+6.47E-01-2.36+5 .321E+02bb

First, +6.47E-01is 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+0200 is read, converted, and placed in location C. Here again, the input number is legitimate and is converted and stored, even though it is not the number desired.

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

Examples:

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.x10^{-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
1.bEb1	E6.3	10.	Blanks are interpreted as zeros

9.4.3 Fw.d OUTPUT

The field occupies w positions in the output record; the corresponding list item must be a floating point quantity that appears as a decimal number, right justified:

ba...a.a...a

The b indicates a blank. The a's represent the most significant digits of the number. The d specifies the number of decimal places to the right of the decimal. If d is zero or omitted, digits to the right of the decimal point are not printed. If the number is positive, the + sign is suppressed. If the field is too short to accommodate the number, asterisks appear in the whole output field. If the field is longer than required to accommodate the number, the number is right justified with blank fill to the left. If the number is out of range, an R is provided. If the number is indefinite, I is printed.

> ANSI FORTRAN,X3.9-1966, does not output any asterisks if field is too narrow or the letters I or R if value is indefinite/out of range.

Contents of A	Format Statement	Print Statement	Printed Result
+32.694	10 FORMAT (F7.3)	PRINT 10,A	b32.694
+32.694	11 FORMAT (F10.3)	PRINT 11,A	bbbb32.694
-32.694	12 FORMAT (F6.3)	PRINT 12,A	****** (no provision for - sign and most significant digit)
.32694	13 FORMAT (F4.3, F6.3)	PRINT 12,A,A	.327b0.327

9.4.4 Fw.d INPUT

On input, the F specification is treated identically to the E specification.

Examples:

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 37925x10^{-7}
-4.7366	F7	-4.7366	No d in specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.2	.62543	Decimal point overrides d of specification
+144.15E-03	F11.2	.14415	Exponents are legitimate in F input and may have P-scaling
5bbbb	F5.2	500.00	No fraction subfield; input number converted as 50000x10^{-2}

9.4.5 Gw.d OUTPUT

The field occupies w positions of the output record, with d significant digits. The real data will be represented by F conversion unless the magnitude of the data exceeds the range that permits effective use of F conversion. In this case, the E conversion will represent the external output. Therefore, the effect of the scale factor is not implemented unless the magnitude of the data requires E conversion.

When F conversion is used under Gw.d output specification, four blanks are inserted within the field, right justified. Therefore, for effective use of F conversion, d must be $\leq$ w-6.

The method of representation in the output record is a function of the magnitude N of the real data being converted. The following table gives a correspondence between N and the method of conversion.

Format:

$0.1 \leq N < 1$	F (w-4).d, 4X
$1 \leq N < 10$	F (w-4).(d-1), 4X
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F (w-4).1, 4X
$10^{d-1} \leq N < 10^{d}$	F (w-4).0, 4X
otherwise	Ew.d or nPEw.d

Examples:

```
        PRINT 101, XYZ          XYZ contains 77.132
101     FORMAT(G10.3)
         Result:  bb77.1bbbb
        PRINT 101, XYZ          XYZ contains 1214635.1
101     FORMAT(G10.3)
         Result:  b.121E+07
```

9.4.6 Gw.d INPUT

Gw.d specification is the same as the Fw.d input specification.

9.4.7 Dw.d OUTPUT

The field occupies w positions of the output record; the list item is a double precision quantity which appears as a decimal number, right justified:

s.a...a±eee	$100 \le eee \le 512$
or	
s.a...aD±ee	$0 \le ee \le 99$

s indicates no character position or minus. D conversion corresponds to Ew.d output.

Single precision numbers cannot be output under Dw.d. If the field specified is too short to accommodate the number, asterisks appear in the whole output field.

> ANSI FTN,X3.9-1966, does not enter asterisks if the field is too narrow.

9.4.8 Dw.d INPUT

D conversion corresponds to E conversion except that the list variables must be double precision names. D is acceptable in place of E as the beginning of an exponent subfield.

Example:

```
        DOUBLE Z,Y,X
        READ1,Z,Y,X
    1   FORMAT (D18.11,D15,D17.4)
```

Input Card:

$$\underbrace{\text{-6.31675298443E-03}}_{18}\underbrace{\text{+2.718926453147}}_{15}\underbrace{\text{6293477528869D-09}}_{17}$$

In storage:

```
    Z = -6.31675298443D-03
    Y =  2.718926453147D+00
    X =  6.293477528869D-01
```

9.4.9 Iw AND Iw.z OUTPUT

The I specification is used to output decimal integer values.

Iw   Iw.z

w is a decimal integer constant designating the total number of characters in the field including signs and blanks. If the integer is positive, the plus sign is suppressed. Numbers in the range of $-2^{59} + 1$ to $2^{59} - 1$ ($2^{59}-1$=576 460 752 303 423 487) are output correctly.

z is a decimal integer constant designating the minimum number of digits output. Leading zeros are generated when the output value requires less than z digits. If z=0, a zero value will produce all blanks. If z=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. Not specifying z produces the same results as z=1.

If the specified field is too short to accommodate the number, asterisks appear in the whole output field.

ANSI FTN,X3.9-1966, does not output asterisks if the field is too narrow.

Example:

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

Result:

$$\underbrace{\text{bbb-3762}}_{8}\underbrace{\text{bbb4762937}}_{10}\underbrace{\text{bbb13}}_{5}$$

9.4.10 Iw AND Iw.z INPUT

The field is w characters in length, and the list item is a decimal integer constant; z is ignored on input. The input field w consists of an integer subfield, containg +, -, 0 through 9, or blank. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. An all blank field is interpreted as minus zero. 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:

$$\underbrace{\text{139}}_{3}\underbrace{\text{bb-15bb}}_{7}\underbrace{\text{18}}_{2}\underbrace{\text{bb7}}_{3}\underbrace{\text{b3}}_{2}\underbrace{\text{b1b4}}_{4}$$

In storage:

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

9.4.11 Ow OR Ow.d OUTPUT

O specification is used to output octal integer values. The output quantity occupies w output record positions right justified:

aa...a

The a's are octal digits. If w is 20 or less, the rightmost w digits appear. If w is greater than 20, the number is right justified in the field with blanks to the left of the output quantity. A negative number is output in its one's complement internal form.

If d is specified, the number is printed with leading zero suppression and with a minus sign for negative numbers. At least d digits will be printed. If the number cannot be output in w octal digits, all asterisks will fill the field.

> Octal output is not specified in ANSI FORTRAN, X3.9-1966.

9.4.12 Ow OR Ow.d INPUT

Octal integer values are converted under O specification. The field is w characters in length, and the list item must be an integer variable.

The input field w consists of an integer subfield only (maximum of 20 octal digits) containing +, -, 0 through 7, or blank.

Only one sign may precede the first digit in the field. Blanks are interpreted as zeros. An all blank field is interpreted as minus zero.

> Octal input is not specified in ANSI FORTRAN, X3.9-1966.

Example:

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

Input Card:

$$\underbrace{3737373737}_{10}\underbrace{666b6644b444}_{12}\underbrace{-0}_{2}$$

In storage:

```
P   00000000003737373737
Q   00000000666066440444
R   77777777777777777777
```

A negative number is represented in one's complement form.

A negative octal number is represented internally in seven's complement form (20 digits) obtained by subtracting each digit of the octal number from seven. For example, if -703 is an input quantity, its internal representation is 77777777777777777074.

```
That is,  77777777777777777777
         -00000000000000000703
          77777777777777777074
```

9.4.13 Aw OUTPUT

The Aw conversion is used to output alphanumeric characters. If w is 10 or more, the quantity appears right justified in the output field, blank fill to left. If w is less than 10, the output quantity is presented by leftmost w characters. If a zero character (6 bits set to zero) occurs, it will be treated as a colon under the 64-character set or as a blank under the 63-character set.

9.4.14 Aw INPUT

This specification accepts FORTRAN characters including blanks. The internal representation is in display code; the field width is w characters.

If w exceeds 10, the input quantity is the rightmost 10 characters in the field. If w is 10 or less, the input quantity is stored as a left justified word; the remaining spaces are blank filled. If a zero character (6 bits set to zero) occurs, it will be treated as a colon under the 64-character set or as a blank under the 63-character set.

Example:

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

Input Card:

$$\underbrace{\text{LUX MENT}}_{8}\underbrace{\text{IS LUX O}}_{8}\underbrace{\text{RBIS}}_{4}$$

In storage:

Q	LUXbMENTbb
P	ISbLUXbObb
O	RBISbbbbbb

9.4.15 Rw OUTPUT

This specification is similar to the Aw output with the following exception: if w is less than 10, the output quantity represents the rightmost characters. If a zero character (6 bits set to zero) occurs, it will be treated as a colon under the 64-character set or as a blank under the 63-character set.

> Rw output is not specified in ANSI FORTRAN, X3.9-1966.

9.4.16 Rw INPUT

This specification is the same as the Aw input with the following exception; if w is less than 10, the input quantity is stored as a right justified binary zero filled word. If a zero character (6 bit set to zero) occurs, it will be treated as a colon under the 64-character set or as a blank under the 63-character set.

> Rw input is not specified in ANSI FORTRAN, X3.9-1966.

Example:

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

Input Card:

```
LUX MENTIS LUX ORBIS
    8       8    4
```

In storage:

Q	00LUXbMENT
P	00ISbLUXbO
O	000000RBIS

9.4.17 Lw OUTPUT

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

A value of TRUE or FALSE in storage causes w-1 blanks followed by a T or F to be output.

Example:

```
      LOGICAL I,J,K,L          I contains -0          J contains 0
      PRINT 5,I,J,K,L          K contains -0          L contains -0
    5 FORMAT (4L3)

       Result:  bbTbbFbbTbbT
```

9.4.18 Lw INPUT

This specification accepts logical quantities as list items. The field is considered true if the first non-blank character in the field is T or false if it is F. An all-blank field is considered false.

9.4.19 Zw INPUT AND OUTPUT

Hexadecimal values are converted under the Z specification.

w is an unsigned integer designating the total number of characters in the field. The input field may contain digits and the letters A through F. A maximum of 15 hexadecimal digits is allowed, blanks and a plus or minus sign may precede the first hexadecimal digit. On output, if w is greater than 15, leading blanks will occur.

> The Zw format is not allowed in ANSI FORTRAN, X3.9-1966.

9.5 nP SCALE FACTOR

The D, E, F, and G conversion may be preceded by a scale factor which is:

External number = Internal number x $10^{\text{scale factor}}$.

A scaled specification is written as shown; n is a signed (positive or negative) integer constant.

nPDw.d   nPEw.d   nPFw.d   nPGw.d   nP   nPEw.dEe   nPEw.dDe

The scale factor is assumed to be zero if no other value has been given; however, once a value has been given, it holds for all D, E, F, and G specifications. To nullify this effect in subsequent D, E, F, and G specifications, a zero scale factor, 0P, must precede a D, E, F, or G specification. Scale factors for D, E, F, and G output specifications must be in the range $-8 \le n \le 8$.

On input, the scale factor has no effect if there is an exponent in the external field. G output makes use of the scale factor only if E conversion is necessary to convert the data. For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

The scaling specification nP may appear independently of a D, E, F, or G specification; it holds for all subsequent D, E, F, and G specifications within the same FORMAT statement unless changed by another nP.

Example:

```
FORMAT(3PE12.6,F10.3,0PD18.7,-1P,F5.2)
```

The E12.6 and F10.3 specifications are scaled by 10^3, the D18.7 specification is not scaled, and the F5.2 specification is scaled by 10^{-1}.

The specification (3P,3I9,F10.2) is the same as the specification (3I9,3PF10.2).

9.5.1 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 314.1592×10^{-2} = 3.141592. However, if an exponent field is read, the scale factor is ignored.

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 3.1415926538 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

9.5.2 Ew.d OR Dw.d SCALING

Output

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

Specification	Output Representation
E20.2	.31E+01
1PE20.2	3.14E+00
2PE20.2	31.42E-01
3PE20.2	314.16E-02
4PE20.2	3141.59E-03
5PE20.2	31415.93E-04
-1PE20.2	.03E+02

9.5.3 Gw.d SCALING

Input

Gw.d scaling on input is the same as Fw.d scaling on input.

Output

The effect of the scale factor is suspended unless the magnitude of the data to be converted is outside the range that permits the effective use of F conversion.

9.6 EDITING SPECIFICATIONS

9.6.1 wX

This specification may be used to include w blanks in an output record or to skip w characters on an input record to permit spacing of input/output quantities. 0X is ignored and X is interpreted as 1X. In the specification list, the comma following X is optional.

If w is negative, then the input/output list is backed up w spaces, but not beyond the first column. On output, any character positions not previously filled during this record generation will be set to blank.

ANSI FORTRAN, X3.9-1966, does not allow 0X or -w.

Examples:

```
        INTEGER A                           A contains 7
        PRINT 10,A,B,C                      B contains 13.6
10      FORMAT(I2,6X,F6.2,6X,E12.5)         C contains 1462.37
          Result: b7bbbbbbb13.60bbbbbbb.14624E+04
        READ 11,R,S,T
11      FORMAT(F5.2,3X,F5.2,6X,F5.2)
        or
11      FORMAT(F5.2,3XF5.2,6XF5.2)
```

Input Card:

```
14.62bb$13.78bCOSTb15.97
```

In storage:

```
R   14.62
S   13.78
T   15.97
```

9.6.2 wH OUTPUT

With this specification, 6-bit characters (including blanks) may be output in the form of comments, titles, and headings. w, an unsigned integer, specifies the number of characters to the right of H that are transmitted to the output record; w may specify a maximum of 136 characters. H denotes a Hollerith field. The comma following the H field is optional.

Examples:

Source program:

```
         PRINT 20
    20   FORMAT (28HbBLANKSbCOUNTbINbANbHbFIELD.)
```

produces output record:

```
    bBLANKSbCOUNTbINbANbHbFIELD.
```

Source program:

```
         PRINT 30,A                    A contains 1.5
    30   FORMAT (6HbLMAX=,F5.2)        Comma optional
```

produces output record:

```
    bLMAX = b1.50
```

9.6.3 wH INPUT

The H specification may be used to read Hollerith characters into an existing H field within the FORMAT specification.

Example:

Source program:

```
         READ 10
    10   FORMAT (27Hbbbbbbbbbbbbbbbbbbbbbbbbbbb)
```

Input Card:

$$\underbrace{\text{bTHIS IS A VARIABLE HEADING}}_{\text{27 cols}}$$

After READ, the FORMAT statement labeled 10 contains the alphanumeric information read from the input card; a subsequent reference to statement 10 in an output statement acts as follows:

```
    PRINT 10
```

produces the print line:

```
    bTHIS IS A VARIABLE HEADING
```

9.6.4 NEW RECORD

The slash(/) signals the end of a record anywhere in the specifications list. Consecutive slashes may appear in a list and they need not be separated from the other list elements by commas. During output, the slash is used to skip lines, cards, or tape records. During input, it specifies that control passes to the next record or card. K(/) or K/ results in K-1 lines being skipped.

Examples:

1.

```
         PRINT 10
      10 FORMAT (6X, 7HHEADING///3X, 5HINPUT, 2X, 6HOUTPUT)
```

Printout:

```
HEADING_________________________line 1
          _____(blank)__________line 2
          _____(blank)__________line 3
INPUTbbOUTPUT___________________line 4
```

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

2.

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

In storage:

A	-11.6
B	.325
C	46.327
D	-14.261

Printout:

```
   -.12E+02bbb.32E+00
  46.327-14.261
```

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

Printout:

```
bb-.12E+02bbb.32E+00_______________________line 1
                          _____blank_______line 2
 b46.327-14.261____________________________line 3
```

4.
```
        PRINT 15, (A(I),I=1,9)
     15 FORMAT (8HbRESULTS2(/) (3F8.2))
```

Printout:

```
 RESULTS___________________________________line 1
                           ___(blank)______line 2
     3.62     -4.03     -9.78______________line 3
    -6.33      7.12      3.49______________line 4
     6.21     -6.74     -1.18______________line 5
```

9.6.5 ≠...≠ SPECIFICATION

The specification ≠...≠ can be used as an alternate form of wH to output headings, titles, and comments. Capabilities are the same as the *...* feature except that ≠ can be enclosed (that is ≠ABC≠≠B≠ is legal and will appear as ABC≠B). The symbol ≠ is the CDC representation of quote marks ("). Refer to appendix A for CDC and ASCII codes and representations.

9.6.6 *...* SPECIFICATION

The specification *...* can be used as an alternate form of wH to output headings, titles, and comments. Any 6-bit character (except asterisk) between the asterisks will be output. The asterisks delineate the Hollerith field. This specification need not be separated from other specifications by commas.

> ANSI FORTRAN, X3.9-1966, does not allow either the *...* or the ≠...≠ specifications.

Output Examples:

1. Source program:

```
        PRINT 10
     10 FORMAT (*bSUBTOTALS*)
```

produces the output record:

```
bSUBTOTALS
```

2. Improper source program to output ABC*BE:

```
       PRINT 1
     1 FORMAT(*ABC*BE*)
```

The * in the output causes the specification to be interpreted as *ABC* and BE*. BE* is an improper specification; therefore, the wH specification must be used to output ABC*BE.

The *...* or ≠...≠ specification can be used to skip alphanumeric data. When a READ occurs, characters in the input stream are skipped and no change is made in the *...* or ≠...≠ specification.

Input Examples:

1. Source program:

```
         READ 10
      10 FORMAT (*bbbbbbbbbbbbbbbbbbbbb*)
```

   Input Card:

```
     FORTRAN FOR THE 7600
```

   A subsequent reference to statement 10 in an output control statement:

```
      PRINT 10
```

   produces:

```
      FORTRAN FOR THE 7600
```

2. Source program:

```
         READ 10
      10 FORMAT (*bbbbbbb*)
```

   Input Card:

```
     HEAD*LINE
```

   A subsequent reference to statement 10 in an output statement:

```
      PRINT 10
```

   Produces:

```
      HEADbbb
```

9.6.7 Tn

This specification is a column selection control.

Tn

n   Unsigned integer. If n = zero, column 1 is assumed.

ANSI FORTRAN X3.9-1966 does not specify Tn.

When Tn is used, control skips columns right or left until column n is reached; then the next format specification is processed. Using card input, if n > 80, the column pointer is moved to column n but a succeeding specification would read only blanks.

```
        READ 40, A,B,C
    40  FORMAT (T1,F5.2, T11,F6.1, T21,F5.2)
```

Input:

```
        84,73bbbbb2436.2bbbb89.14
```

A is set to 84.73, B to 2436.2, and C to 89.14.

```
        WRITE (31,10)
    10 FORMAT (T20,*LABELS*)
```

The first 19 characters of the output record are skipped and the next six characters (LABELS) are written on output unit number 31, beginning in character position 20.

With T specification, the order of a list need not be the same as the printed page or card input. The same information can be read more than once.

When a T specification causes control to pass over character positions on output, those positions not previously filled during this record generation are set to blanks; while those already filled are left unchanged.

9.7 REPEATED FORMAT SPECIFICATION

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

For example, to print two quantities K, L:

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

Specifications for K, L are identical; the FORMAT statement may also be:

```
    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))

Nesting of parenthetical groups preceded by repeat constants beyond two levels is not permitted in FORMAT specifications.

9.8 UNLIMITED GROUPS

Unlimited group repeat is implemented according to the ANSI X3.9 specification. An innermost parenthetical group that has no repeat count specified in a FORMAT statement assumes a group repeat count of one. If the last outer right parenthesis for the FORMAT specification is encountered and the I/O list is not exhausted, control reverts to that group repeat specification terminated by the last preceding right parenthesis, or, if none exists, then to the first left parenthesis of the FORMAT statement.

9.9 VARIABLE FORMAT

FORMAT specifications may be specified at the time of program execution. The specification, including left and right parentheses but not the statement label or the word FORMAT, is read under A conversion or in a DATA statement and stored in an array or a simple variable. The name of the array containing the specifications may be used in place of the FORMAT statement labels in the associated I/O operation. The array name that appears with or without subscripts specifies the location of the first word of the FORMAT information.

> ANSI FORTRAN,X3.9-1966, specifies that only an array name without subscripts may be used in the place of the FORMAT statement label. An nH field description may not be part of a format specification within an array.

Examples:

1. Assume the following FORMAT specifications:

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

This information can be punched in an input card and read by the statements of the program.

```
     DIMENSION IVAR(3)
     READ 1 (IVAR(I),I=1,3)
1    FORMAT (3A10)
```

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

IVAR(1):	(E12.2,F8.
IVAR(2):	2,I7,2E20.
IVAR(3):	3,F9.3,I4)

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

```
     PRINT IVAR, A, B, I, C, D, E, J
```

This produces 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 LAIS1(3),  LAIS2(2),A(6),LSN(3),TEMP(3)
      DATA LAIS1/21H(2F6.3,I7,2E12.2,3I1)/,LAIS2/20H
      (I6,6X,3F4.1,2E12.2)/
```

Output statement:

```
      PRINT LAIS1,(A(I),I=1,2),K,B,C,(LSN(J),J=1,3
```

is the same as:

```
      PRINT 1,(A(I),I=1,2),D,B,C,(LSN(J),J=1,3)
1     FORMAT (2F6.3,I7,2E12.2,3II)
```

Output statement:

```
      PRINT LAIS2,LA,(A(M),M=3,4),A(6),(TEMP(I),I=2,3)
```

is the same as:

```
      PRINT 2,LA,(A(M),M=3,4),A(6),(TEMP(L),L=2,3)
2     FORMAT (I6,6X,3F4.1,2E12.2)
```

3.

```
      DIMENSION LAIS (3),  VALUE(6)
      DATA LAIS/26H(I3,13HMEANbVALUEbIS,F6.3)/
```

Output statement:

```
      WRITE (10,LAIS)NUM,VALUE(6)
```

is the same as:

```
      WRITE (10,10)NUM,VALUE(6)
10    FORMAT (I3,13HMEANbVALUEbIS,F6.3)
```

9.10 VARIABLE SPECIFICATIONS

V and = variables can be used in a FORMAT statement.

The V specification can be used instead of the standard specifications A, D, E, F, G, I, L, O, P, R, T, X, or Z. When V is encountered, the right-most character (bits 0 through 5) from the next variable in the I/O list are substituted for V. The character must be one of the specifications listed above. However, V cannot be used in Ew.dVe for the D or E specification.

The = character may be used in a FORMAT specification whenever a number could be used. The next list item is used as a signed integer value for the number designated by =.

ANSI FORTRAN, X3.9-1966, does not allow the V or = variables.

INPUT/OUTPUT STATEMENTS 10

The following defintions apply to all I/O statements.

i	logical I/O unit number: an integer constant of one or two digits (the first must not be zero) integer variable name of no more than 6 characters, with a value of 1 to 99
n	FORTRAN declaration identifier: statement number variable identifier which references the starting storage location of FORMAT information
L	I/O list

10.1 OUTPUT STATEMENTS

PRINT n, L or
PRINT n

Information in the list (L) is transferred from the storage locations to the standard output unit as line printer images, 136 characters or less per line in accordance with the FORMAT declaration, n.

ANSI FORTRAN, X3.9-1966, does not specify the PRINT statement.

The maximum record length is 136 characters, but the first character of every record is not printed as it is used for carriage control when printing on-line. Characters in excess of the print line are truncated. Each new record starts a new print line.

Character	Action
Blank	Single-space before printing
0	Double-space before printing
1	Eject page before printing
+	Suppress spacing before printing; print two successive records on the same line

Consult the operating system manual for additional characters.

PUNCH n, L or

PUNCH n

Information is transferred from the storage locations given by the list (L) identifiers to the standard punch unit. Information is transferred as Hollerith images, 80 characters or less per card in accordance with the FORMAT declaration, n. Records greater than 80 characters will be truncated.

ANSI FORTRAN, X3. 9-1966, does not specify PUNCH statement.

WRITE (i, n)L

This statement transfers information from storage locations given by the list (L) to a specified output unit (i) according to the FORMAT declaration (n).

WRITE (i)L

This statement transfers information from storage locations given by the list (L) to a specified output unit (i). If L is omitted, the WRITE (i) statement acts as a do-nothing statement. The list written by this statement constitutes one binary record. See READ (i)L.

ANSI FORTRAN, X3. 9-1966, does not provide a WRITE(i) without list.

Examples:

```
1.          DIMENSION A(260), B(4000)
            WRITE(10)A, B

2.          DO 5 I=1, 10
       5    WRITE 6, AMAX (I), (M(I, J), J=1, 5)

3.          PRINT 50, A, B, C, (I, J)
      50    FORMAT (X8HMINIMUM=F17. 7, 2X8HMAXIMUM=F17. 7, 2X10HVALUE IS $F8. 2)

4.          PRINT 51, (A(I), I=1, 20)
      51    FORMAT(X23HTRUTH MATRIX VALUES ARE/(3X4L3))

5.          DIMENSION LISTNME(2), FSTNME(2)
            PUNCH 52, ACCT, LSTNME, FSTNME, TELNO, SHPDTE, ITMNO
      52    FORMAT (F8. 2, 3X4A10, XI5)

6.          WRITE (2, 52)A, B, C, D
      53    FORMAT (4E21. 9)

7.          WRITE (2, 53)A, B, C, D

8.          WRITE (2, 54)
      54    FORMAT (32HTHIS STATEMENT HAS NO DATA LIST.)
```

10.2 READ STATEMENTS

Check for the end of the file either by counting records, checking for a predefined termination record, or by checking for an IF EOF statement after each read (paragraph 10.4). If an EOF is read on a formatted read (READ n,list, or READ(i,n)list) the data used for processing will be a blank. If an EOF is read on a binary read (READ(i)list) the data named in the list will not be changed by the read.. If a read is issued after the EOF is read, the job will be terminated unless the EOF flag has been cleared by an IF EOF statement. An EOS encountered when reading a BCD file is ignored except for file INPUT where it is treated as an EOP.

Format:

```
    READ n,list
```

One or more card images are read from the standard input unit. Information is converted from left to right in accordance with FORMAT specification (n), and it is stored in the locations named by the list.

ANSI FORTRAN, X3.9-1966, does not specify the READ n,list statement.

Example:

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

READ(i,n)list

This statement transfers one logical record of information from logical unit (i) to storage locations named by the list, according to FORMAT specification (n). The number of words in the list and the FORMAT specifications must conform with the record structure on the logical unit.

READ(i)list

This statement transfers one logical binary record of information from a specified unit (i) to storage locations named by the list.

Records to be read by READ (i) should be written in binary mode. The number of words in the list of READ (i) list must not exceed the number of words in the corresponding WRITE statement.

If list is omitted, READ (i) spaces over one logical record. See WRITE (i)list.

Examples:

```
1.          DIMENSION C(264)
            DIMENSION BMAX (10),M2(10,5)
            READ (10)C
            DO 7 I=1,10
       7    READ(0)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 (0) ((A(I,J),I=1,100),J=1,50)
            DIMENSION Z(8)

2.          DOUBLE PRECISION DB(4)
            READ (10,51)DB
      51    FORMAT (4D20.12)
            READ 51,DB
            READ (2,52) (A(J),J=1,8)
      52    FORMAT (F10.4)
```

10.3 NAMELIST STATEMENTS

The NAMELIST statement permits the input and output of character strings consisting of names and values without a format specification.

Format:

NAMELIST $/y_1/a_1/y_2/a_2/\ldots/y_n/a_n$

Each y is a NAMELIST name consisting of 1-7 characters which must be unique within the program unit in which it is used. Each a is a list of the form $b_1, b_2, \ldots b_n$; each being a variable or array name.

In any given NAMELIST statement, the list a of variable names or array names between the NAMELIST identifier y and the next NAMELIST identifier (or the end of the statement if no NAMELIST identifier follows) is associated with the identifier y; that is, the list a_i is associated with NAMELIST identifier y_i.

Examples:

```
PROGRAM MAIN
NAMELIST/NAME1/N1,N2,R1,R2/NAME2/N3,R3,N4,N1
SUBROUTINE XTRACT (A,B,C)
NAMELIST/CALL1/L1,L2,L3/CALL2/L3,P4,L5,B
```

A variable name or array name may be an element of more than one such list. In a subprogram, b may be a dummy parameter identifying a variable or an array, but the array may not have variable dimensions.

A NAMELIST name may be defined only once in a program unit preceding any reference to it. Once defined, any reference to a NAMELIST name may be made only in a READ or WRITE statement. The form of the I/O statements used with NAMELIST is as follows:

```
READ (u,x)
WRITE (u,x)
```

u is an integer variable or integer constant denoting a logical unit, and x is a NAMELIST name.

ANSI FORTRAN, X3.9-1966, does not specify the NAMELIST statement nor the READ (u,x) WRITE (u,x) statement.

Example:

```
Assume A, I, and L are array names
        .
        .
        .
NAMELIST /NAM1/A,B,I,J/NAM2/C,K,L
        .
        .
        .
READ (5,NAM1)
        .
        .
        .
WRITE (8,NAM2)
```

These statements result in BCD I/O on the device specified as the logical unit of the variables and arrays associated with the identifiers, NAM1 and NAM2.

Input Data

The current file on unit u is scanned up to an end of file or a record with a $ in column 2 followed immediately by the name (NAM1) with no embedded blanks. Succeeding data items are read until a $ is encountered.

The data item, separated by commas, may be in any of three forms:

v=c

a=$d_1, \dots, d_j$

a(n)=$d_1, \dots, d_m$

v is a variable name, c a constant, a an array name, and n is an integer constant subscript. d_i are simple constants or repeated constants of the form k*c, where k is the repetition factor.

Example:

```
DIMENSION Y(3.5)
LOGICAL L
COMPLEX Z
NAMELIST /HURRY/I1,I2,I3,K,M,Y,Z,L
READ (5,HURRY)
```

and the input record:

```
$HURRYbI1=+1,L=.TRUE.,I2=2,I3=3.5,Y(3.5)=26,Y(1,1)=11,12.0E1,13,4*14,
Z=(1.,2.),K=16,M=17$
```

produces the following values:

```
I1=1              Y(1,2)=14.0
I2=2              Y(2,2)=14.0
I3=3              Y(3.2)=14.0
Y(3,5)=26.0       Y(1,3)=14.0
Y(1,1)=11.0       K=16
Y(2,1)=120.0      M=17
Y(3,1)=13.0       Z=(1.,2.)
                  L = .TRUE
```

The number of constants, including repetitions, given for an unsubscripted array name must equal the number of elements in that array. For a subscripted array name, the number of constants need not equal, but may not exceed, the number of array elements needed to fill the array.

v=c — variable v is set to c

$a=d_1,\ldots,d_j$ — the values $d_1,\ldots,d_j$ are stored in consecutive elements of array a in the order in which the array is stored internally.

$a(n)=d_1,\ldots,d_m$ — elements are filled consecutively starting at a (n)

The specified constant of the NAMELIST statement may be integer, real, double precision, complex of the form (c_1, c_2), or logical of the form .T., or .TRUE., .F., or .FALSE.. A logical or complex variable may be set only to a logical and complex constant, respectively. Any other variable may be set to an integer, real or double precision constant. Such a constant is converted to the type of its associated variable.

Constants and repeated constant fields may not include embedded blanks. Blanks, however, may appear elsewhere in data records.

A maximum of 150 characters per input record is permitted. More than one record may be used for input data. All except the last record must end with a constant followed by a comma, and no serial numbers may appear; the first column of each record is ignored.

The set of data items may consist of any subset of the variable names associated with x, the NAMELIST name. These names need not be in the order in which they appear in the defining NAMELIST statement.

Output Data

The NAMELIST statement produces BCD output to unit u as follows:

> One record consisting of a $ in column 2 immediately followed by the identifier x. As many records as are needed to output the current values of all variables in the list associated with x. Simple variables are output as v=c.
>
> Elements of dimensioned variables are output in the order in which they are stored internally (by columns).
>
> The data fields are made large enough to include all significant digits. Logical constants appear as T and F. No data appears in column 1 of any record.
>
> One record consisting of a $ in column 2 is immediately followed by the letters END.
>
> The records output by such a WRITE statement may be read by a READ (u, x) statement where x is the same NAMELIST identifier.
>
> If unit u is the standard punch unit and a record is longer than 80 characters, the remaining characters are punched on the next card.
>
> The maximum length of a record written by a WRITE (u, x) statement is 130 characters.

10.4 FILE HANDLING STATEMENTS

REWIND i

File unit i is repositioned to the beginning of information. If the file is already rewound, the statement acts as a do-nothing statement. The REWIND statement cannot reference the system I/O files.

BACKSPACE i

File unit i is backspaced one record in an unformatted file or a BUFFER IN/BUFFER OUT file or one unit record in a formatted BCD file. If unit i is at the beginning of information, this statement acts as a do-nothing statement. The BACKSPACE statement must not reference the system I/O files.

END FILE i

An end-of-partition is written on file unit i. The ENDFILE statement may not reference the system I/O files.

IF (ENDFILE i)n_1, n_2

IF (EOF, i)n_1, n_2

These statements check the previous read operation to determine whether an end-of-partition has been encountered on file tape i. If so, 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

n_1	Not ready
n_2	Ready and no previous error
n_3	End of partition sensed on last operation
n_4	Parity error sensed on last operation

This statement must be used following a BUFFER IN/BUFFER OUT operation.

ANSI FORTRAN, X3.9-1966, does not specify the IF (ENDFILE i)n_1, n_2 IF(EOF, i)n_1, n_2 IF(UNIT, i)n_1, n_2, n_3, n_4 statements.

10.5 BUFFER STATEMENTS

The primary differences between BUFFER I/O and READ/WRITE I/O statements are:

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. The parity mode must remain constant for a particular file.

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 to perform other tasks while data transmission is in progress. Before buffered data is used, the status of the buffer operation should be checked. A read/write control statement completes the operation before returning control to the program.

The IF (UNIT, i)n_1, n_2, n_3, n_4 statement must be used to check the status of a BUFFER IN/BUFFER OUT operation.

In the descriptions that follow, these definitions apply.

u	Logical unit number
p	Parity key. May be specified by a constant or simple variable (not subscripted). O for even parity (coded characters); non-zero for odd parity.
a	Variable identifier: First word of data block to be transmitted.
b	Variables identifier: Last word of data block to be transmitted.

In the BUFFER statements the address of b must be greater than that of a.

BUFFER IN (i,p) (a,b)

Information is transmitted from unit i to storage locations a through b. The transmission will be terminated when either:

1. All the data from a to b has been read in, the file will be left positioned at the end of the record.

2. The end of the current record is encountered.

In either case, the number of words actually transmitted can be obtained with:

L=LENGTH(i)

The use of this statement is described in the BUFFEI description in Appendix J.

BUFFER OUT (i,p) (a,b)

Information is transmitted from storage locations a through b as one logical record. It is written on unit i containing all the words from a to b inclusive. The use of this statement is shown under BUFFEO in Appendix J.

ANSI FORTRAN, X3.9-1966, does not specify the BUFFER statement.

Examples:

1.
```
      COMMON/BUFF/DATA(10),CAL(50)
      PAR=0
      BUFFER IN(9,PAR) (DATA(1),CAL(50))
    5 IF(UNIT,9)5,6,7,8
```

Information is input from unit 9 to the labeled common area BUFF beginning at DATA(1), the first word of the buffer and extending through CAL(50), the last word of the buffer.

2.
```
      DIMENSION A(100)
      N=6
      BUFFER OUT(n,1)(A(I),A(100))
    4 IF(UNIT,n)4,6,7,8
```

Information is transmitted to unit n from the buffer area defined by A(1) and A(100); that is, all of array A is transmitted. If unit n refers to magnetic tape, data will be written in odd parity.

10.6 ENCODE/DECODE STATEMENTS

The ENCODE/DECODE statements are comparable to the formatted WRITE/READ statements; however, no peripheral equipment is involved. Information is transferred under FORMAT specifications from one area of storage to another. The parameters in these statements are defined as follows:

ENCODE (c, n, v)list

c Unsigned integer constant or a simple integer variable (not subscripted) specifying the number of characters in the record. c may be an arbitrary number of BCD characters

n Statement number, variable identifier, or formal parameter representing the FORMAT statement

v Variable identifier or an array identifier which supplies the starting location of the BCD record

list I/O list

When encoding or decoding is performed, the first record begins with the leftmost character position specified by v and continues c BCD characters (10 BCD characters per computer word). For ENCODE, if c is not a multiple of 10, the record ends in the middle of a word and the remainder of the word is blank filled. For DECODE, if the record ends with a partial word the balance of the word is ignored.

Since each succeeding record begins with a new computer word, an integral number of computer words is allocated for each record with (c+9)/10 words. There is no intrinsic limit on C unless V is a level 2 variable, then c must be 150 or less.

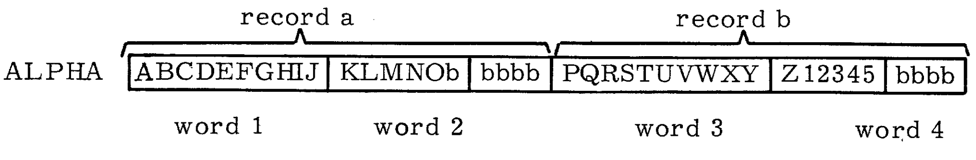
> ANSI FORTRAN, X3.9-1966, does not specify the ENCODE/DECODE statements.

Examples:

```
A(1) = ABCDEFGHIJ
A(2) = KLMNObbbbb
B(1) = PQRSTUVWXY
B(2) = Z12345bbbb
```

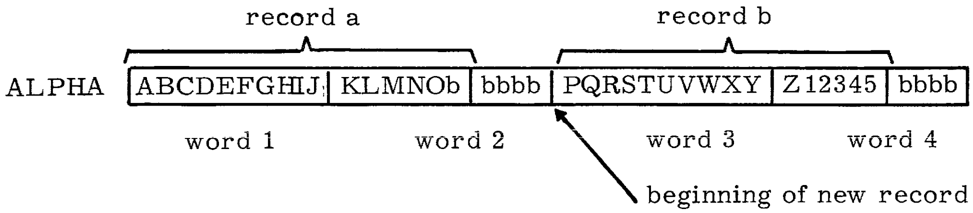
1. c = multiple of 10

```
     DIMENSION A(2), B(2)
     ENCODE (20, 1, ALPHA)A, B
1    FORMAT (A10, A5/A10, A6)
```



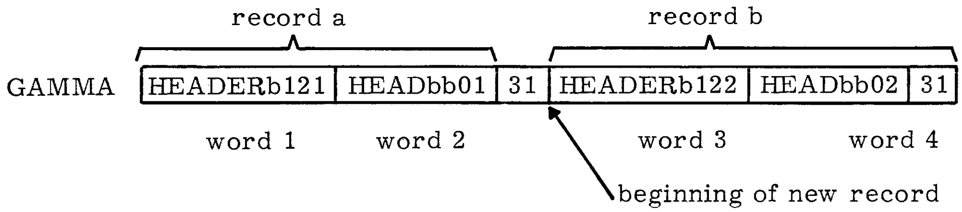
2. c ≠ multiple of 10

```
     DIMENSION A(2),B(2)
     ENCODE (16,1,ALPHA)A,B
   1 FORMAT (A10,A6)
```



3. c ≠ multiple of 10

```
     DIMENSION A6(2),B6(2)
     DECODE (18,1,GAMMA)A6,B6
   1 FORMAT (A10,A8)
```



A6(1) = HEADERb121

A6(2) = HEADbb01bb

B6(1) = HEADERb122

B6(2) = HEADbb02bb

ENCODE (c,n,v)list

The information of the list variables, list, is transmitted according to the FORMAT (n) and stored in locations starting at v, c characters per record. If the I/O list (list) and specification list (n) translate more than c characters per record, an execution diagnostic occurs. If the number of characters converted is less than c, the remainder of the record is filled with blanks.

DECODE (c, n, v)list

The information in c consecutive characters (starting at address v) is transmitted according to the FORMAT n and stored in the list variables. If the number of characters specified by the I/O list and the specification list (n) is greater than c (record length) per record, an execution diagnostic occurs. If DECODE attempts to process an illegal code or a character illegal under a given conversion specification, an execution diagnostic occurs.

If a DECODE statement encounters a zero character (6 bits all clear) while processing an A or R format, that character will be treated as a colon under a 64-character set or as a blank under a 63-character set.

Examples:

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

```
    LOC(1) SSSSSxxxxx
          .
          .
          .
    LOC(6) xxxxxddddd
          10 ch/wd
```

   To form SSSSSddddd in storage location NAME:

```
    DECODE (10,1,LOC(6))TEMP
1    FORMAT (5X,A5)
    ENCODE (10,2,NAME)LOC(1),TEMP
2    FORMAT(2A5)
```

   The DECODE statement places the last 5 characters of LOC(6) into the first 5 characters of TEMP. The ENCODE statement packs the first 5 characters of LOC(1) and TEMP into NAME.

   With the R specification; the program may be shortened to:

```
    ENCODE (10,1,NAME)LOC(1),LOC(6)
1    FORMAT (A5,R5)
```

2. ENCODE 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 $2 \le m \le 9$. The following program permits m to vary.

   Format:

```
    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 ensure that 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: (A8,Im). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A8,Im).

A and B will be printed 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 rearrange and change the information in a record. The following example also illustrates that it is possible to encode an area into itself and that encoding will destroy information previously contained in an area.

```
          I = 10HV = bbFT/SEC
          IA = 16
          ENCODE (10,1,I)I,IA,I
      1   FORMAT (A2,I2,R6)
```

Before executing the above code:

```
    I = 26545555062450230503
```

After execution:

```
    I = 26543441062450230503
```

STANDARD SCOPE CHARACTER SETS A

The character set selected when the system is installed should be compatible with the printers.

With an installation parameter, the installation keypunch format standard can be selected as 026 or 029; the installation parameter can also allow a user to override the standard; a user may select a keypunch mode for his input deck by punching 26 or 29 in columns 79 and 80 of his JOB card or any 7/8/9 end-of-record card. The mode remains set for the remainder of the job or until it is reset by a different mode selection on another 7/8/9 card.

TABLE A-1. CDC 64-CHARACTER SET

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	: †	8-2	8-2	00††	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	-	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	%	8-6	0-8-4	16
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	∨	11-0 or	11-0 or	52
27	W	0-6	0-6	26			11-8-2	11-8-2	
30	X	0-7	0-7	27	67	∧	0-8-7	12	37
31	Y	0-8	0-8	30	70	↑	11-8-5	8-4	55
32	Z	0-9	0-9	31	71	↓	11-8-6	0-8-7	56
33	0	0	0	12	72	<	12-0 or	12-0 or	72
34	1	1	1	01			12-8-2	12-8-2	
35	2	2	2	02	73	>	11-8-7	0-8-6	57
36	3	3	3	03	74	≤	8-5	12-8-4	15
37	4	4	4	04	75	≥	12-8-5	0-8-2	75
					76	¬	12-8-6	11-8-7	76
					77	; (semicolon)	12-8-7	11-8-6	77

TABLE A-2. ASCII 64-CHARACTER SUBSET†

Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code	Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code
00	:††	8-2	8-2	072	40	5	5	5	065
01	A	12-1	12-1	101	41	6	6	6	066
02	B	12-2	12-2	102	42	7	7	7	067
03	C	12-3	12-3	103	43	8	8	8	070
04	D	12-4	12-4	104	44	9	9	9	071
05	F	12-5	12-5	105	45	+	12	12-8-6	053
06	F	12-6	12-6	106	46	-	11	11	055
07	G	12-7	12-7	107	47	*	11-8-4	11-8-4	052
10	H	12-8	12-8	110	50	/	0-1	0-1	057
11	I	12-9	12-9	111	51	(	0-8-4	12-8-5	050
12	J	11-1	11-1	112	52	)	12-8-4	11-8-5	051
13	K	11-2	11-2	113	53	$	11-8-3	11-8-3	044
14	L	11-3	11-3	114	54	=	8-3	8-6	075
15	M	11-4	11-4	115	55	blank	no punch	no punch	040
16	N	11-5	11-5	116	56	, (comma)	0-8-3	0-8-3	054
17	O	11-6	11-6	117	57	. (period)	12-8-3	12-8-3	056
20	P	11-7	11-7	120	60	≠	0-8-6	8-3	043
21	Q	11-8	11-8	121	61	' (apostrophe)	8-7	8-5	047
22	R	11-9	11-9	122	62	!	0-8-2	12-8-7	041
23	S	0-2	0-2	123	63	%	8-6	0-8-4	045
24	T	0-3	0-3	124	64	" (quote)	8-4	8-7	042
25	U	0-4	0-4	125	65	_(underline)	0-8-5	0-8-5	137
26	V	0-5	0-5	126	66	-]	11-0 or	11-0 or	175
27	W	0-6	0-6	127			11-8-2	11-8-2	
30	X	0-7	0-7	130	67	&	0-8-7	12	046
31	Y	0-8	0-8	131	70	@	11-8-5	8-4	100
32	Z	0-9	0-9	132	71	?	11-8-6	0-8-7	077
33	0	0	0	060	72	[	12-0 or	12-0 or	173
34	1	1	1	061			12-8-2	12-8-2	
35	2	2	2	062	73	>	11-8-7	0-8-6	076
36	3	3	3	063	74	<	8-5	12-8-4	074
37	4	4	4	064	75	\	12-8-5	0-8-2	134
					76	(circum-	12-8-6	11-8-7	136
						flex)			
					77	; (semicolon)	12-8-7	11-8-6	073

† BCD representation is used when data is recorded on even parity magnetic tape. In this case, the octal BCD/display code correspondence is the same as for the CDC 64-character set.

†† This character is lost on even parity magnetic tape.

TABLE A-3. CDC 63-CHARACTER SET †

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	(none)†			16	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	-	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	: (colon)†	8-2	8-2	00 ††
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	∧	11-0 or 11-8-2	11-0 or 11-8-2	52
27	W	0-6	0-6	26	67		0-8-7	12	37
30	X	0-7	0-7	27	70	↑	11-8-5	8-4	55
31	Y	0-8	0-8	30	71	↓	11-8-6	0-8-7	56
32	Z	0-9	0-9	31	72	<	12-0 or 12-8-2	12-0 or 12-8-2	72
33	0	0	0	12	73	>	11-8-7	0-8-6	57
34	1	1	1	01	74	≤	8-5	12-8-4	15
35	2	2	2	02	75	≥	12-8-5	0-8-2	75
36	3	3	3	03	76		12-8-6	11-8-7	76
37	4	4	4	04	77	; (semicolon)	12-8-7	11-8-6	77

† When the 63-character set is used, the punch code 8-2 is associated with display code 63, the colon. Display code 00_8 is not included in the 63-character set and is not associated with any card punch. The 8-6 card punch (026 keypunch) and the 0-8-4 card punch (029 keypunch) in the 63-character set are treated as blank on input.

†† Since 00 cannot be represented on magnetic tape, it is converted to BCD 12. On input, it will be translated to display code 33 (number zero).

FORTRAN STATEMENT LIST B

SUBPROGRAM STATEMENTS

Entry Points		PROGRAM name ($f_1,\ldots,f_n$)†	N
		FORTRAN II PROGRAM name ($f_1,\ldots,f_n$) †	N
		SUBROUTINE name ($p_1,\ldots,p_n$)	N
		FORTRAN II SUBROUTINE name ($p_1,\ldots,p_n$)†	N
		FUNCTION name ($p_1,\ldots,p_n$)	N
		type FUNCTION name ($p_1,\ldots,p_n$)	N
		FORTRAN II FUNCTION ($p_1,\ldots,p_n$)†	N
		FORTRAN II type FUNCTION ($p_1,\ldots,p_n$)†	N
		ENTRY name †	N
Intersubroutine		EXTERNAL $name_1, name_2\ldots$ †	N
Transfer Statements	F††	$name_1, name_2, \ldots$	N
		CALL name	E
		CALL name ($p_1,\ldots,p_n$)	E
		RETURN	E

DATA DECLARATION AND STORAGE ALLOCATION

Type Declaration	COMPLEX list	N
	DOUBLE PRECISION list †	N
	DOUBLE list†	N
	REAL list	N
	INTEGER list	N
	LOGICAL list	N
	TYPE DOUBLE list†	N
	TYPE COMPLEX list†	N
	TYPE REAL list†	N
	TYPE INTEGER list†	N
	TYPE LOGICAL list†	N

N = Nonexecutable E = Executable

† Non ANSI

†† Column 1 indicates F is used in FORTRAN II mode, non ANSI.

Storage Allocations	DIMENSION $v_1, v_2, \dots, v_n$		N
	COMMON/I_1/$list_1$/I_2/$list_2$... /I_n/$list_n$		N
	EQUIVALENCE (A,B,...),(A1,B1,...)...		N
	DATA I_1/list/,I_2/list/,...		N
	DATA (I_1=list),(I_2=list),...		N
	BLOCK DATA n		N
	LEVEL n,list †		N
ARITHMETIC STATEMENT FUNCTION			
	name ($p_1, p_2, \dots, p_n$) = Expression		E
SYMBOL MANIPULATION, CONTROL AND I/O			
Replacement	A=E	Arithmetic	E
	L=E	Logical/Relational	E
	M=E	Masking	E
Intraprogram Transfers	GO TO k		E
	GO TO m		E
	GO TO m, ($n_1, n_2, \dots, n_m$)		E
	GO TO ($n_1, n_2, \dots, n_m$)i		E
	IF (c)n_1, n_2, n_3		E
	IF (e)n_1, n_2 †		E
	IF (ℓ)n_1, n_2		E
	IF (ℓ)s		E
	IF (SENSE LIGHT i)n_1, n_2 †		E
	IF (SENSE SWITCH i)n_1, n_2 †		E
	IF DIVIDE CHECK n_1, n_2 †		E
	IF (ENDFILE i)n_1, n_2 †		E
	IF (EOF, i)n_1, n_2 †		E
	IF (UNIT, i)n_1, n_2, n_3, n_4 †		E
	IF ACCUMULATOR OVERFLOW n_1, n_2 †		E
	IF QUOTIENT OVERFLOW n_1, n_2 †		E
LOOP CONTROL	DO n i = m_1, m_2, m_3		E
	DO n i = m_1, m_2		E

† Non-ANSI

MISCELLANEOUS PROGRAM CONTROLS

ASSIGN kTO m	E
SENSE LIGHT i	E
CONTINUE	E
PAUSE	E
PAUSE n	E
STOP	E
STOP n	E

I/O FORMAT

FORMAT (spec$_1$, . . . , k(spec$_m$1, . . .), spec$_n$, . . .)	N
NAMELIST /y$_1$/a$_1$/y$_2$/a$_2$/. . . /y$_n$/a$_n$	N

I/O CONTROL STATEMENTS

	READ n, L	E
	READ TAPE i, L †	E
	READ INPUT TAPE i, n, L †	E
	PRINT n, L	E
	PUNCH n, L	E
	READ (i, n)L	E
	WRITE (i, n)L	E
	WRITE TAPE i, L †	E
	WRITE OUTPUT TAPE i, n, L †	E
	READ (i)L	E
	WRITE (i)L	E
	ENCODE (c, n, v)L †	E
	DECODE (c, n, v)L †	E
	BUFFER IN (u, p)(A, B) †	E
	BUFFER OUT (u, p)(A, B) †	E
I/O File Handling	END FILE i	E
	REWIND i	E
	BACKSPACE i	E

PROGRAM AND SUBPROGRAM TERMINATION

END	E
END name †	E

† Non ANSI

LIBRARY SUBPROGRAMS

Intrinsic Function and No. of Arguments	Definition	Example	Symbolic Name	Type of Argument	Type of Function
				Type of	
Absolute value (1)	[a]	Y=ABS(X)	ABS	Real	Real
		J=IABS(I)	IABS	Integer	Integer
Truncation (1)	Trunc (a) = [a] if $a \geq 0$ (a) = -[a] if $a < 0$ where the function [a] is defined to be the integer i that satisfies $i \leq a < i+1$	Y=AINT(X)	AINT	Real	Real
		I=INT(X)	INT	Real	Integer
Modulo	MOD or AMOD (a_1, a_2) is defined to be a_1-trunc(a_1, a_2)*a_2	B=AMOD(A1,A2)	AMOD	Real	Real
		J=MOD(I1,I2)	MOD	Integer	Integer
Choosing largest value ($\geq$ 2)	Max $(a_1, a_2, \ldots)$	X=AMAX0(I,J,K)	AMAX0	Integer	Real
		A=AMAX1(X,Y,Z)	AMAX1	Real	Real
		L=MAX0(I,J,K,N)	MAX0	Integer	Integer
		I=MAX1(A,B)	MAX1	Real	Integer
		DOUBLE W,X,Y,Z W=DMAX1(X,Y,Z)	DMAX1	Double	Double
Choosing smallest value ($\geq$ 2)	Min $(a_1, a_2, \ldots)$	Y=AMIN0(I,J)	AMIN0	Integer	Real
		Z=AMIN1(X,Y)	AMIN1	Real	Real
		L=MIN0(I,J,K)	MIN0	Integer	Integer
		J=MIN1(X,Y)	MIN1	Real	Integer
		DOUBLE A,B,C C=DMIN1(A,B)	DMIN1	Double	Double
Float (1)	Conversion from integer to real	XI=FLOAT(I)	FLOAT	Integer	Real
Fix (1)	Conversion from real to integer	IY=IFIX(Y)	IFIX	Real	Integer
Transfer of sign (2)	Sign of a_2 times a_1	Z=SIGN(X,Y)	SIGN	Real	Real
		J=ISIGN(I1,I2)	ISIGN	Integer	Integer
Positive difference (2)	a_1 - Min(a_1, a_2)	Z=DIM(X,Y)	DIM	Real	Real
		J=IDIM(I1,I2)	IDIM	Integer	Integer
Obtain real part of complex argument (1)		COMPLEX A B=REAL(A)	REAL	Complex	Real
Obtain imaginary part of complex argument (1)		D=AIMAG(A)	AIMAG	Complex	Real
Express two real arguments in complex form (2)	$a_1 + a_2$ -1	COMPLEX C C=CMPLX(A1,A2)	CMPLX	Real	Complex
Obtain conjugate of a Complex argument (1)		COMPLEX X,Y Y=CONJG(X)	CONJG	Complex	Complex

LIBRARY SUBPROGRAMS (Cont'd)

Intrinsic Function and No. of Arguments	Definition	Example	Symbolic Name	Type of	
				Argument	Function
Logical product (2)	$a_1 \wedge a_2$	C=AND(A1,A2)	AND	Single word	Octal
Logical sum (2)	$a_1 \vee a_2$	D=OR(A1,A2)	OR	Single word	Octal
Complement (1)	$\bar{a}$	B=COMPL(A)	COMPL	Single word	Octal

External Function and No. or Arguments	Defintion	Example	Symbolic Name	Type of	
				Argument	Function
Exponential (1)	e^a	Z=EXP(Y)	EXP	Real	Real
		DOUBLE X,Y Y=DEXP(X)	DEXP	Double	Double
		COMPLEX A,B B=CEXP(A)	CEXP	Complex	Complex
Natural logarithm (1)	$\log_e(a)$	Z=ALOG(Y)	ALOG	Real	Real
		Y=DLOG(X)	DLOG	Double	Double
		B=CLOG(A)	CLOG	Complex	Complex
Common logarithm (1)	$\log_{10}(a)$	B=ALOG10(A)	ALOG10	Real	Real
		DD=DLOG10(D)	DLOG10	Double	Double
Trigonometric sine (1)	sin (a)	Y=SIN(X)	SIN	Real	Real
		DS=DSIN(D)	DSIN	Double	Double
		CS=CSIN(C)	CSIN	Complex	Complex
Trigonometric cosine (1)	cos(a)	X=COS(Y)	COS	Real	Real
		DC=DCOS(D)	DCOS	Double	Double
		CC=CCOS(C)	CCOS	Complex	Complex
Hyperbolic tangent (1)	tanh (a)	B=TANH(A)	TANH	Real	Real
Square root (1)	$(a)^{1/2}$	Y=SQRT(X)	SQRT	Real	Real
		DY=DSQRT(DX)	DSQRT	Double	Double
		CY=CSQRT(CX)	CSQRT	Complex	Complex
Arctangent (1)	arctan (a)	Y=ATAN(X)	ATAN	Real	Real
		DY=DATAN(DX)	DATAN	Double	Double
(2)	arctan (a_1/a_2)	B=ATAN2(A1,A2)	ATAN2	Real	Real
		D=DATAN2(D1,D2)	DATAN2	Double	Double
Modulus (1)	$\text{AIMAG}^2(a)+\text{REAL}^2(a)$	CM=CABS(CX)	CABS	Complex	Real
Arccosine (1)	arccos (a)	X=ACOS(Y)	ACOS	Real	Real
Arcsine (1)	arcsin (a)	X=ASIN(Y)	ASIN	Real	Real
Trigonometric tangent (1)	tan (a)	Y=TAN(X)	TAN	Real	Real

LIBRARY SUBPROGRAMS (Cont'd)

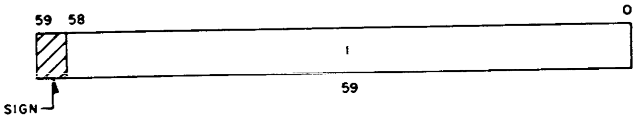
External Functions			Symbolic	Type of	
and No. of Arguments	Defintion	Example	Name	Argument	Function
Address of argument (1)	loc (a)	P=LOCF(X)	LOCF	Symbolic	Integer
Length (1)	Number of central memory words read on the previous I/O request for a particular file	L=LENGTH(J)	LENGTH	Integer	Integer
Variable characteristic (1)	-1 = indefinite +1 = out of range 0 = Normal	LEN=LEGVAR(V)	LEGVAR	Real	Integer
Parity status on non-buffer unit (1)	0 = no parity error on previous read	IP=IOCHEC(5)	IOCHEC	Integer	Integer
Date as returned by SCOPE (1)	date(a)	WHEN=DATE(D)	DATE†	Value Returned	Hollerith
Current reading of system clock as returned by SCOPE (1)	time(a)	CLTIM=TIME(A)	TIME†	Variable	Hollerith
Time in seconds (1)	second (a) (accumulated CP time)	CLTM=SECOND(A)	SECOND	Real	Real
Absolute value (1)	a	DOUBLE A, B B=DABS(A)	DABS††	Double	Double
Truncation (1)	Trunc (a) = [a] if $a \geq 0$ (a) = -[a] if $a < 0$ where the function [a] is defined to be the integer i that satisfies $i \leq a < i+1$	DOUBLE Z J=IDINT(Z)	IDINT††	Double	Integer
Modulo (2)	a_1-trunc(a_1, a_2)*a_a	DM=DMOD(D1, D2)	DMOD	Double	Double
Transfer of sign (2)	Sign of a_2 times a_1	D3=DSIGN(D1, D2)	DSIGN††	Double	Double
Truncate to obtain most significant part of double precision argument (1)		DOUBLE Y X=SNGL(Y)	SNGL	Double	Real
Express single precision argument in double precision form (1)		DOUBLE Y Y=DBLE(X)	DBLE††	Real	Double
Random number generator Each call returns a random number in the interval (0, 1).		Y=RANF(X)	RANF	Dummy	Real

† These routines may be used as functions or subroutines. The value is always returned via the argument and via the normal function return.

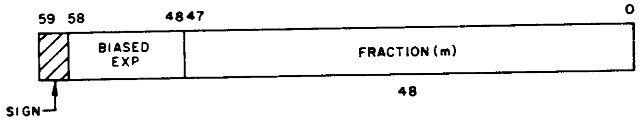
ANSI FORTRAN, X3.9-1966, specifies these functions as intrinsic.

COMPUTER WORD STRUCTURE OF CONSTANTS

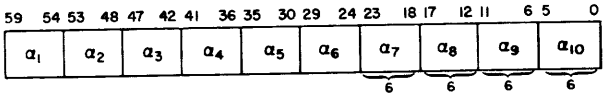
INTEGER



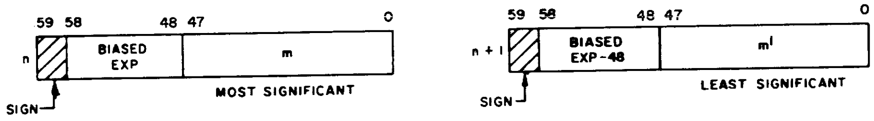
REAL



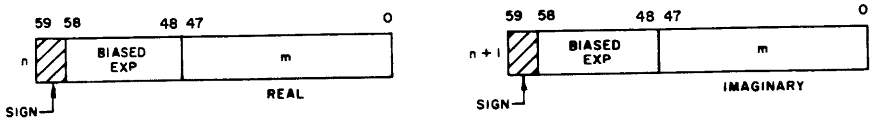
HOLLERITH BCD AND DISPLAY CODE



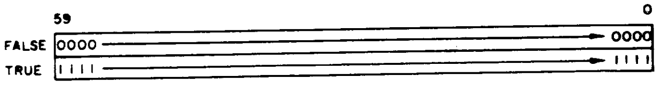
DOUBLE-PRECISION



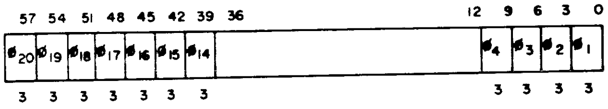
COMPLEX



LOGICAL



OCTAL



COMPILATION AND EXECUTION E

The CONTROL DATA CYBER 70/Model 76 FORTRAN RUN compiler may be called into execution by using either of two control statement formats. The parameters on one format are in fixed positions as used in previous RUN compilers. The second form is free form and includes a rounded floating point option not available in the fixed format.

FIXED PARAMETER POSITION FORTRAN CONTROL STATEMENT

```
RUN(cm,,,if,of,bf,lc,,cs,t,el,lm)
```

cm Compiler mode option (if omitted, assume G)

- G Compile and execute, no list unless explicit LIST cards appear in the deck
- S Compile with source list, no execute
- P† Compile with source list and punch deck on file PUNCHB, no execute
- L Compile with source and object list which contains mnemonics, no execute
- M† Compile with source and object list which contains mnemonics, produce a punch deck on file PUNCHB, no execute

if File name for compiler input; if omitted, assumed to be INPUT

of File name for compiler output; if omitted, assumed to be OUTPUT

bf File name on which the binary information is always written; if omitted, assumed to be LGO

lc Line-limit (octal) on the OUTPUT file of an object program; if omitted, assumed to be 10000_8. Line limit may be altered at execution time as a parameter on EXECUTE or load-and-execute control card. Example: LGO(LC=2000). The LC parameter may appear anywhere in the parameter list of the execution control card. It is not counted as a file name for file equivalencing purposes. The value used is interpreted as an octal number

cs Cross-reference switch. If non-zero, a cross reference listing is produced

t Error traceback. When this parameter is present, calls to math library functions will be made with maximum error checking. Full error traceback will be done if an error is detected except for those functions whose names are declared in an EXTERNAL statement. For these functions, no error traceback is done. When t is omitted, no error checking will be done on math library functions other than EXP, ALOG, ALOG10, SIN, COS, ATAN, and ATAN2; no traceback will be provided if errors are encountered. Thus, a significant saving in memory space and execution time is realized. This mode of compilation (t omitted) is not intended for use with programs in the debug stages

el Forced load switch. If other than zero or blank, generated code is forced-loaded, despite any errors which may have been generated during compilation

lm LCM address mode code. If present, 21-bit mode is used; otherwise, 17-bit mode is used

† Because COMPASS allows only one binary output file to be written, a RUN (P or M) and LGO will result in only the FORTRAN code of a FORTRAN-COMPASS job being placed on LGO.

The second, third, and eighth positions are allowed so that 6000 compatibility is maintained. If used, they are ignored by the 7600 Compiler.

LCM Address Mode

If 17-bit mode is selected, the LCM object time field length is restricted to a maximum of 131072 words. The 17-bit option allows generation of more efficient object code, using 18-bit address field instructions and index registers to hold addresses and perform index arithmetic. If 21-bit address mode is selected, the LCM object time field length is limited only by the amount of large core memory available or 2097152 words, whichever is smaller; however, this will tend to produce a less efficient object code which utilizes indirect address words in small core memory. In either mode the size of any one array or common block is restricted to a maximum of 131072 words.

FREE FORM FORTRAN CONTROL CARD

RUN(p_1, p_2, p_3, ..., p_n)

Where the parameter list may contain any of the following parameters.

1. Compile mode parameter

   This parameter may consist of any combination of the characters C, R, X, and T with one of the characters G, S, P, L, or M.

   The latter are as defined for the fixed format (if omitted, G is assumed) and

   C selects the cross reference table option.

   R selects the rounded floating point arithmetic option, otherwise unrounded floating point instructions are used.

   X selects the execute mode option. This mode will cause the compiler to terminate normally even when fatal errors have occurred.

   T selects traceback to check arguments for math functions.

2. Number of print lines parameter

   NLx where x is an octal number (x=99999) that specifies the maximum number of print lines. If omitted, NL10000 is assumed.

3. Source input file parameter

   I=fn where fn is the file name for the compiler input. If omitted, I=INPUT is assumed.

4. Listing output file parameter

   O=fn where fn is the file name for compiler listing output. If omitted, O=OUTPUT is assumed.

5. Binary output file parameter

   B=fn where fn is the file name for compiler binary output. If omitted, B=LGO is assumed.

6. LCM address mode parameter

   LCM=D
   or
   LCM=I

   The D specification specifies 17-bit LCM address mode. The I specification specifies 21-bit LCM address mode. If this parameter is omitted, 17-bit mode is assumed.

As an example, the call card RUN (LRC,O=PRINTF) will cause the compiler to generate a full octal listing, generate rounded floating point operations, generate a cross reference table, read its input from file INPUT, write its output on file PRINTF, and write its binary output on file LGO. If fatal errors occur during compilation, the compiler will abort the job.

COMPILATION FEATURES

Source and Object Code Output

Compiler output, except in the G mode, includes a reproduction of the source program, a variable map, and indications of fatal and non-fatal errors detected during compilation. If the G mode is selected, all output is suppressed unless the LIST control card is used. If the L or M mode is selected, the output includes an object list which contains mnemonics.

On L or M mode listings, the following lines appear:

PS------(preamble start)

PT------(preamble terminate)

CS------(conclusion start)

CT------(conclusion terminate)

These identify statement sequences where some common computation has been extracted and performed before entering the sequence.

A copy of the compiled programs is always left in disk storage as binary record on a file named either LGO or the name specified as the bf parameter in the call to the compiler. The compiled program may be called and executed repeatedly, until the end of job occurs, by using the name of the load-and-go file. In the output file at the end of compilation of each subprogram, the compiler indicates the amount of unused compilation space.

Two control cards LIST and NOLIST are available to allow the programmer more flexibility in requesting a list of his programs. These cards are free field beyond column 6 and appear between subprograms. When the LIST card is detected, the source cards of the following programs are listed. If the compiler mode was L, the object code is also listed. When the NOLIST card is detected no more listing takes place until a LIST card is detected.

Overlay Files

To aid in the preparation of overlay files, the FORTRAN compiler, upon detecting an OVERLAY card between subprograms, transfers them to the load-and-go file, and to the PUNCHB file if the P or M option is selected. They also are transferred to the output file.

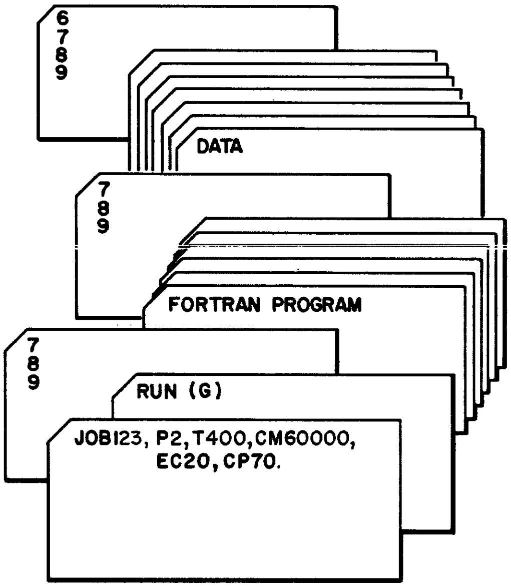
The following control card is transferred to the load-and-go and PUNCHB file if mode is P or M:

OVERLAY (lfn,l_1,l_2)

This statement must begin after column 6.

SAMPLE JOB DECKS

Compile and Execute

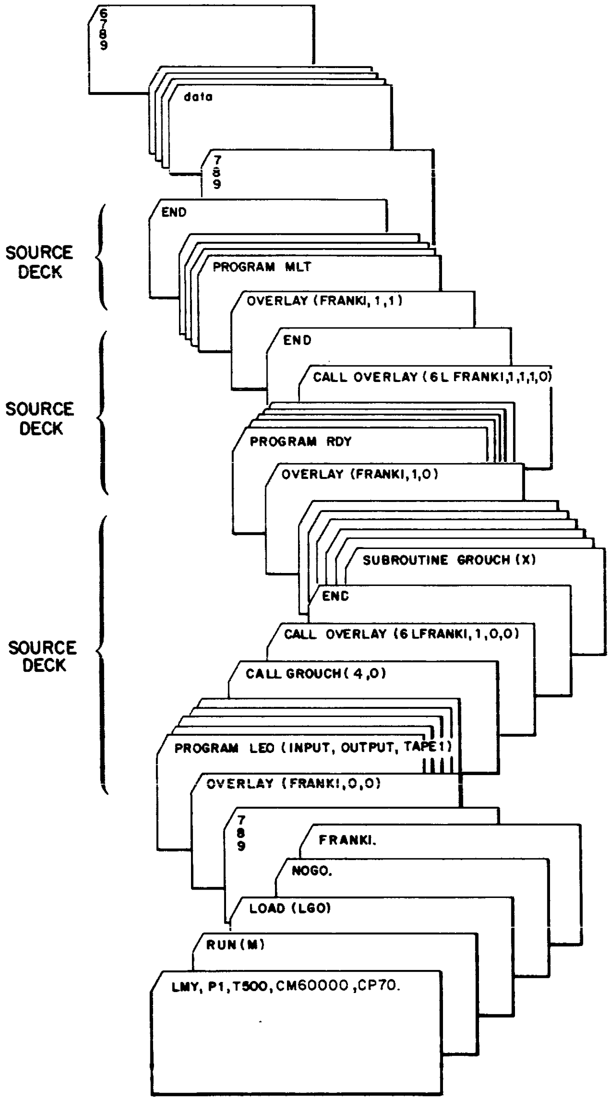


Job name	JOB123
Priority	2
Time limit	Approximately 4 minutes
SCM field length	60000_8 words
LCM field length	20000_8 words

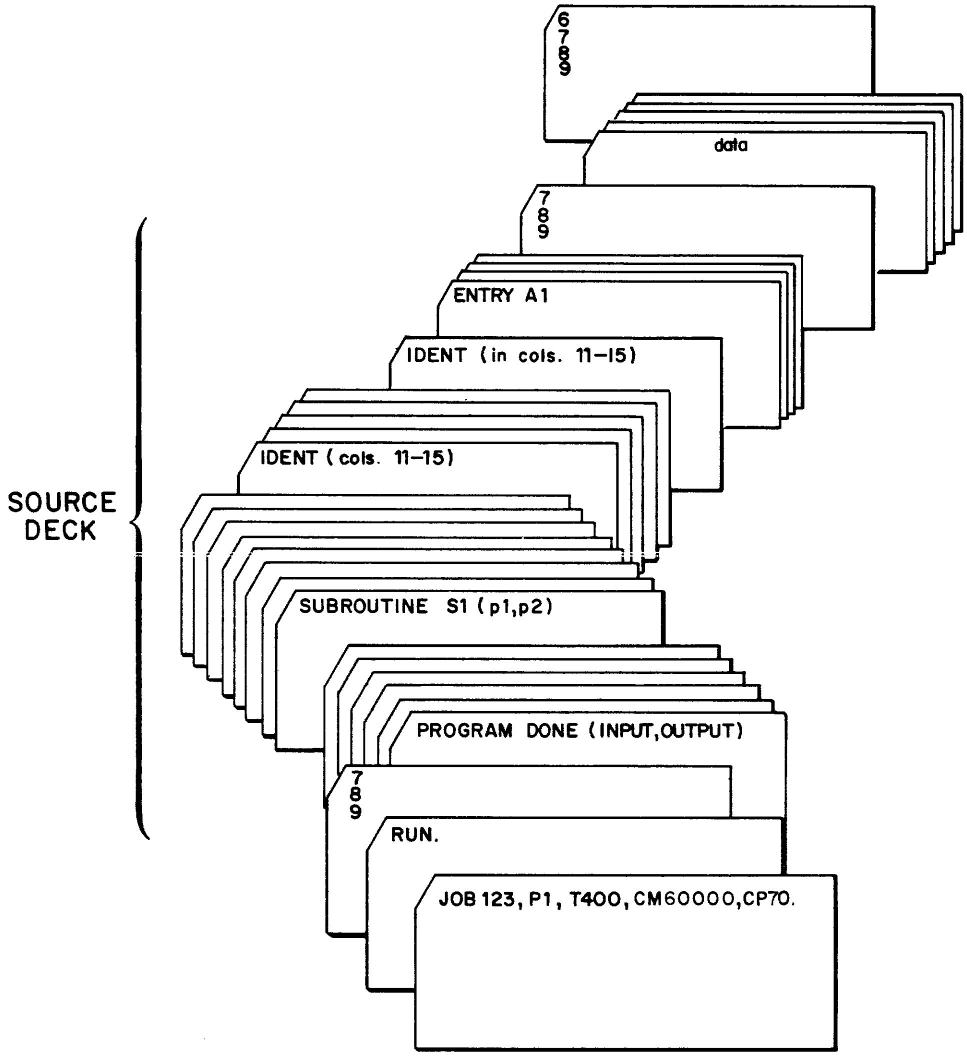
Compile and execute with no list and no binary deck.

The above control card sequence will compile and run in a SCM field of 60000_8 words.

Overlay Preparation of 0, 0;1, 0;1, 1



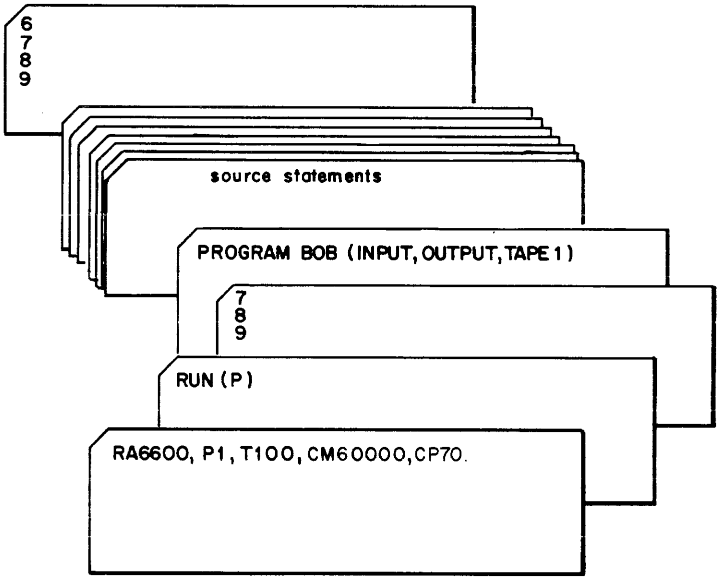
FORTRAN Compile and Execute with Mixed FORTRAN and COMPASS Subprograms



FORTRAN Compile and Produce Binary Card; no execution.

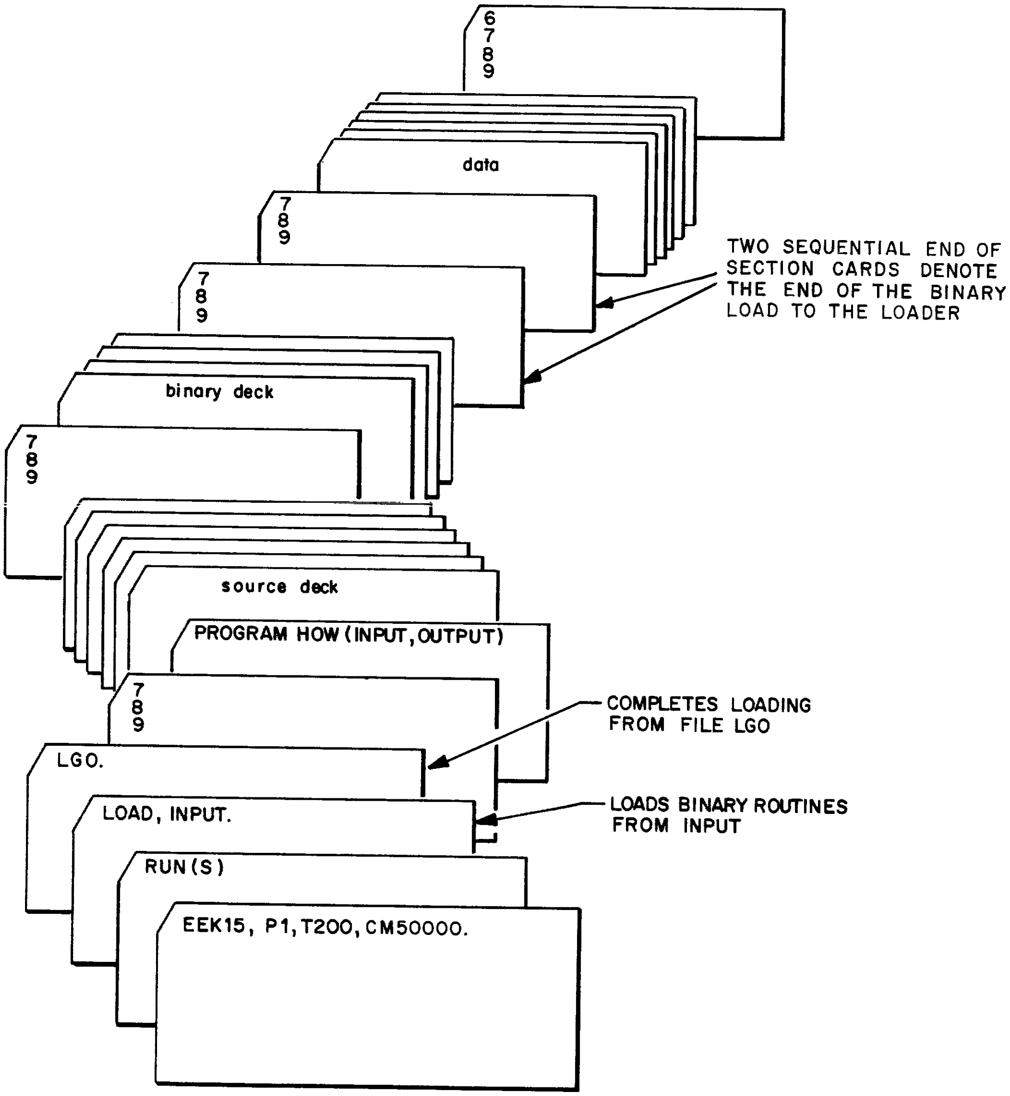
Three files of I/O - INPUT, OUTPUT AND TAPE1

Job name	RA6600
Priority	1
Time limit	approximately 1 minute
Field length	50000_8 words


6
7
8
9
source statements
PROGRAM BOB (INPUT, OUTPUT, TAPE 1)
7
8
9
RUN (P)
RA6600, P1, T100, CM60000, CP70.

FORTRAN Compile and Execute (plus a prepunched binary subroutine deck)

A binary deck to be loaded with a compiled routine must be preceded by 7/8/9 card.



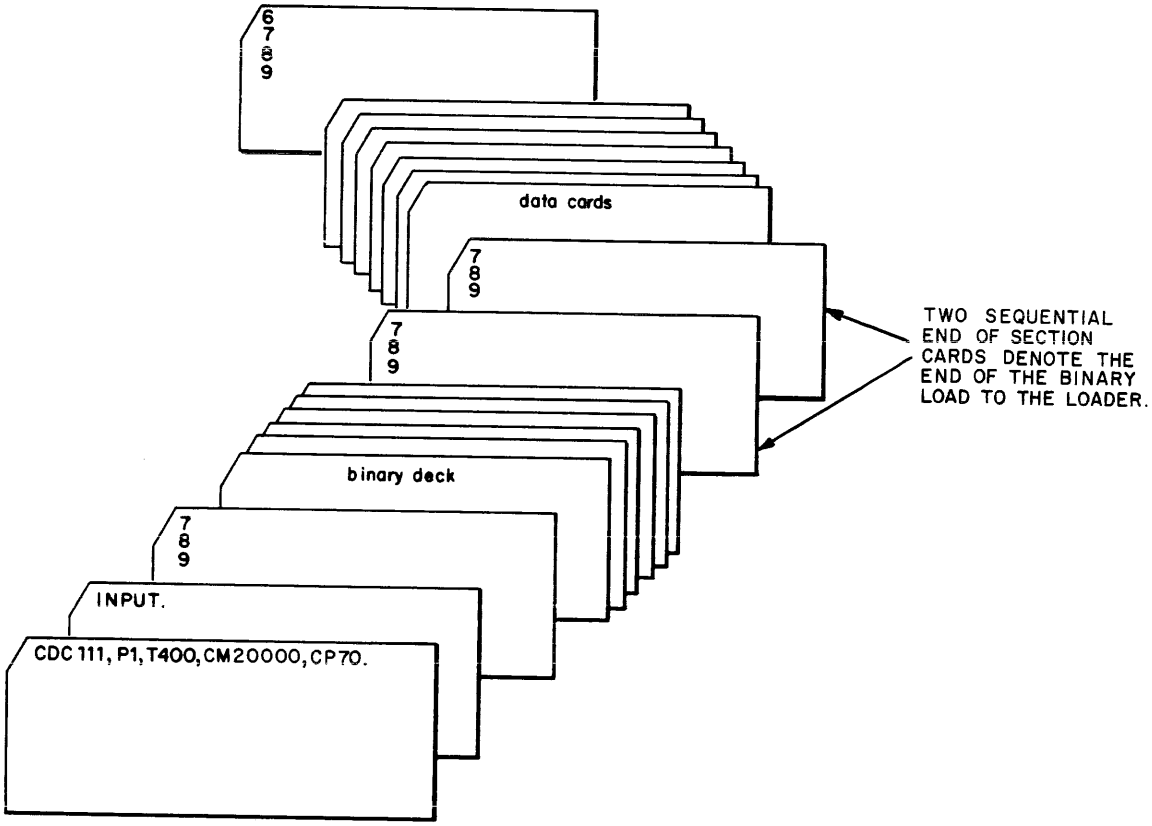
TWO SEQUENTIAL END OF SECTION CARDS DENOTE THE END OF THE BINARY LOAD TO THE LOADER

COMPLETES LOADING FROM FILE LGO

LOADS BINARY ROUTINES FROM INPUT

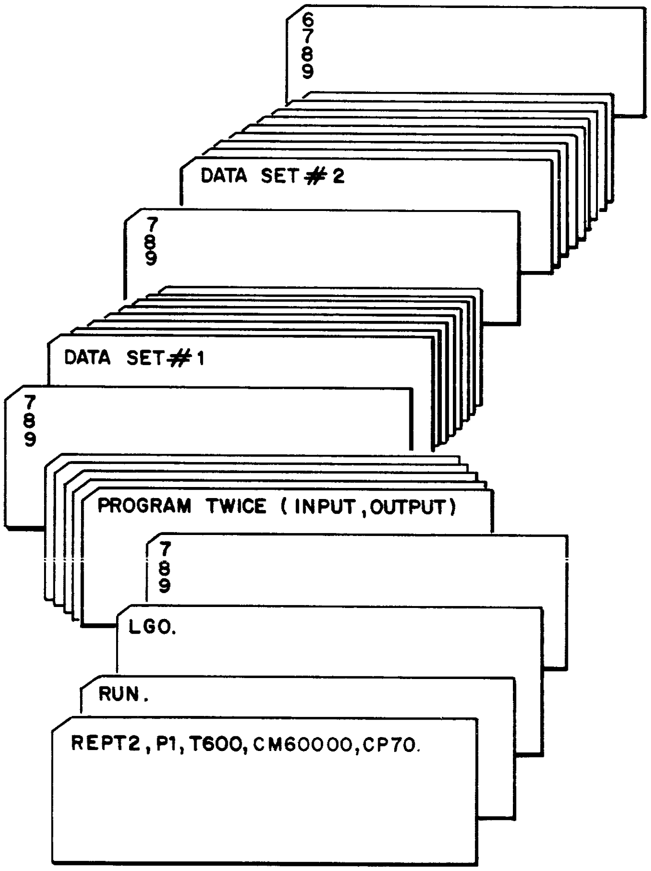
Load and Execute a Prepunched Binary Program

The binary cards in the input file following the record separator are loaded into central memory when the program call card INPUT is encountered.



6
7
8
9
data cards
7
8
9
7
8
9
binary deck
7
8
9
INPUT.
CDC 111, P1,T400,CM20000,CP70.
TWO SEQUENTIAL END OF SECTION CARDS DENOTE THE END OF THE BINARY LOAD TO THE LOADER.

Compile Once and Execute Twice† with Different Data Decks


6
7
8
9
DATA SET # 2
7
8
9
DATA SET # 1
7
8
9
PROGRAM TWICE (INPUT, OUTPUT)
7
8
9
LGO.
RUN.
REPT2, P1, T600, CM60000, CP70.

† PROGRAM TWICE must read the end-of-record card.

PROGRAM-SUBPROGRAM FORMAT

The starting address of all programs is RA+100_8 with the first 77_8 locations containing file and loader information. As many as 50 files may be declared for any one program. An object time routine, Q8NTRY, stores the file names along with their FIT addresses in locations RA+2 through RA+1+n, where n is the number of files declared on the PROGRAM card. The I/O buffers are allocated and managed by 7000 DATA MANAGER in LCM.

The first word of a main program is a trace word containing the name of the program in left justified complemented display code and the argument count +100_8 in the lower 18 bits. The program name is complemented so that an object time traceback routine can distinguish between RUN and FORTRAN Extended trace words. The second word of a main program is the entry point. It contains instructions to preset the parameters for Q8NTRY which performs initialization only once per execution. Therefore, entry into an overlay is through this word destroying its contents. Since Q8NTRY does not perform any function after the first entry, the destruction of the preset parameters for an overlay entry does not matter.

The first n words of a subroutine are reserved for argument addresses for the subroutines where:

n   max (0, argument count-6) in 17-bit LCM mode
     or, the argument count in 21-bit LCM mode

In 17-bit LCM mode, one word is reserved for each argument in excess of six. The addresses of the first six arguments are passed through B registers 1 through 6. In 21-bit LCM mode one word is reserved for each argument. In this mode the address of argument i is passed through B register i and the reserved word is unused if the argument is in SCM and $i \leq 6$. Otherwise, the argument address is passed through the ith reserved word.

Immediately following these reserved words is the subroutine trace word containing the name of the subroutine in left justified complemented display code and the argument count in the lower six bits. Next is the entry/exit line for the subroutine. Therefore, a subroutine will have two reserved words if the argument count is zero.

Subroutines written in the COMPASS assembly language that will operate in conjunction with FORTRAN coded routines should be formatted as in the following examples to take full advantage of the error tracing facility of FORTRAN Version 2.0. The called subroutines do not have to be concerned with register preservation.

Example:

PROGRAM PETE (INPUT, OUTPUT, TAPE1)

DATA 0	L00002	Complement of name; argument count plus 100_8
SB1 L00002	L00001	Entry/Exit line
SB2 C00001		Where C00001 is the address of an argument list for Q8NTRY.
RJ Q8NTRY	L00003	

```
SUBROUTINE PHD  (A,B,C)

     DATA  0           L00005        Reserved for A†
     DATA  0           L00004        Reserved for B†
     DATA  0           L00003        Reserved for C†
     DATA  0           L00002        Name and argument count (trace word)
     DATA  0           L00001        Entry/Exit line

SUBROUTINE PEN  (A,B,C,D,E,F,G,H,I,J)

     DATA  0           L00003        Reserved for A†
     DATA  0           L00004        Reserved for B†
     DATA  0           L00005        Reserved for C†
     DATA  0           L00006        Reserved for D†
     DATA  0           L00007        Reserved for E†
     DATA  0           L00010        Reserved for F†
     DATA  0           L00011        Reserved for G
     DATA  0           L00012        Reserved for H
     DATA  0           L00013        Reserved for I
     DATA  0           L00014        Reserved for J
     DATA  0           L00002        Name and argument count (trace word)
     DATA  0           L00001        Entry/Exit line
```

Calling Sequence to PEN (17-bit LCM mode)

```
CALLPEN (M,N,O,P,Q,R,S,T,U,V)

     SB1    M
     SB2    N
     SB3    O
     SB4    P
     SB5    Q
     SB6    R
     SX6    Entry line of PEN
SA   SA1    X6-1     Name and argument count
     SB7    X1-6     Number of arguments less 6
     SX6    S
     SA6    A1-B7    Reserved word for S
     SX7    T
     SA7    A6+1     Reserved word for T
     SX6    U
     SA6    A7+1     Reserved word for U
     SX7    V
     SA7    A6+1     Reserved word for V
     RJ     PEN
     0712L00002      Where 12$_8$ is argument count and L00002 is the trace word containing
                     the name of calling routine.
```

† These words do not appear if 17-bit LCM mode is requested on the RUN control card.

SYSTEM ROUTINE G

The SYSTEM routine handles error tracing, diagnostic printing, termination of output buffers, and transfer to specified nonstandard error procedures. All the FORTRAN mathematical routines rely on SYSTEM to complete these tasks. Also, a FORTRAN coded routine may call SYSTEM. Any of the parameters used by SYSTEM relating to a specific error may be changed by a user routine during execution. The END processor also makes use of SYSTEM to dump the output buffers and print an error summary. Since the initialization routine (Q8NTRY.), the end processors (END., STOP., and EXIT.), and SYSTEM, must always be available, these routines are combined into one subprogram with multiple entry points.

The calling sequence to SYSTEM passes the error number as the first parameter and an error message as the second parameter. Several different messages may be associated with one error number. The error summary given at program termination lists the total number of times each error number was encountered.

The error number of 0 is accepted as a special call to end the output buffers and return. If no OUTPUT file is defined before SYSTEM is called, no errors are printed and a message to this effect appears in the dayfile. Each printed line is subjected to the line limit of the OUTPUT buffer; when the limit is exceeded,the job is terminated.

The error table is ordered serially (the first error corresponds to error number 1) and it is expandable at assembly time. The last entry in the table is a catch-all for any error number which exceeds the table length. An entry in the error table appears as follows:

Print Frequency	Print Frequency Increment	Print Limit	Error Detection Total	F/ NF	A/ NA	Non-Standard Recovery Address
8	8	12	12	1	1	18

Print Frequency = PF

Print Frequency Increment = PFI

PF = 0 and PFI = 0	The diagnostic and traceback information are not listed
PF = 0 and PFI = 1	The diagnostic and traceback information are listed until the print limit is reached
PF = 0 and PFI = n	The diagnostic and traceback information are listed only the first n times unless the print limit is reached first
PF = n	The diagnostic and traceback information are listed n^{th} time until the print limit is reached

Fatal (F)/Nonfatal (NF)

If the error is nonfatal and a nonstandard recovery address is not specified, error messages are printed according to PRINT FREQUENCY and control is returned to the calling routine.

If the error is fatal and no nonstandard recovery address is specified, error messages are printed according to PRINT FREQUENCY, an error summary is listed, all output buffers are terminated, and the job is terminated.

If a nonstandard recovery address is specified, see Nonstandard Recovery.

Nonstandard Recovery

SYSTEM supplies the nonstandard recovery routine with the following information:

B1-B6 Address of the first six parameters passed to the routine that detected the error

X1 Error number passed to SYSTEM

X2 Address of the diagnostic message available to SYSTEM

X3 Address within an auxiliary table if A/NA bit is set

X4 Instruction word consisting of the return jump to SYSTEM in the upper 30 bits and trace back information in the lower 30 bits for the routine which detected the error

A0 Address of error number entry within SYSTEM's error table

Nonfatal Error

The entry/exit line of the routine which called SYSTEM is set into the entry/exit line of the recovery routine. Control is then passed to the word immediately following the entry/exit line of the recovery routine. The traceback information available to SYSTEM from the routine which detected the error is passed to the recovery routine in X4.

Any faulty parameters may be corrected, and the recovery routine is allowed to call the routine which detected the error with corrected parameters. Upon exit from the recovery routine, control is turned not to SYSTEM nor to the routine which detected the error, but rather back another level (see example). By not correcting the faulty parameters in the recovery routine, a three routine loop could develop between the routine which detects the error, SYSTEM, and the recovery routine. No checking is done for this case.

Example:

```
          MAIN
E/E

          .
          .
          .
          CALL MATH (A,B,C)
RTN1      .              Point of return from MATH, if no errors are detected, or
          .              from RECOVERY
          .
          END
```

```
            MATH
E/E         jump to RTN1              May be reentered from RECOVERY with
                                      corrected parameters
            .
            .
            .
            RJ SYSTEM
            07XXAAAAAA.........       Traceback information
RTN2        .
            .
            .
            END

            SYSTEM
            jump to RTN2              Transfers E/E line of MATH to E/E line of
            .                         RECOVERY and gives control to RECOVERY
            .
            .
            JUMP TO RECOVERY
            .
            .
            .
            END

            RECOVERY
E/E         jump to RTN1              Corrects faulty parameters and may recall
                                      MATH
            .
            .
            .
            RJ MATH
            .
            .
            .
            jump to E/E               Returns to MAIN following reference to
            END                       MATH
```

Fatal Error

SYSTEM calls the nonstandard recovery routine in the normal fashion, with the registers set as indicated in the preceding chart. If the nonstandard recovery routine exits in the normal fashion returning control to SYSTEM, an error summary is listed, all output buffers are terminated, and the job is terminated.

Use of the A/NA Bit

The A/NA bit is used only when a nonstandard recovery address is specified.

If this bit is set, the address within an auxiliary table is passed in the third word of the secondary parameter list to the recovery routine. This bit allows more information than is normally supplied by SYSTEM to be passed to the recovery routine. The bit may be set only during assembly of SYSTEM, as an entry must also be made into the auxiliary table. Each word in the auxiliary table must have the error number in its upper 10 bits so that the address of the first error number match is passed to the recovery routine. An entry in the auxiliary table for an error is not limited to any specific number of words.

The traceback information is terminated as soon as one of the following conditions is detected:

The calling routine is a program

The maximum traceback limit is reached

No traceback information is supplied

To change an error table during execution, a FORTRAN type call is made to SYSTEMC with the following parameters:

Error Number

List containing the consecutive locations:

Word 1	Fatal/nonfatal (fatal = 1, nonfatal = 0)
Word 2	Print frequency
Word 3	Print frequency increment (only significant if word 2 - 0) special values:

word 2 = 0, word 3 = 0	Never list error
word 2 = 0, word 3 = 1	Always list error
word 2 = 0, word 3 = X	List error only the first X times

Word 4	Print limit
Word 5	Nonstandard recovery address
Word 6	Maximum traceback limit

If any word within the parameter list is negative, the value already in table entry is not to be altered.

(Since auxiliary table bit may be set only during assembly of SYSTEM, only then can an auxiliary table entry be made.)

Error Listing

Message supplied by calling routine:

```
    ERROR NUMBER xxxx DETECTED BY zzzzzzz AT yyyyyy
    CALLED FROM cccccc AT ADDRESS wwwwww
        or
    CALLED FROM cccccc AT LINE dddd
```

zzzzzzz and cccccc are routine names, yyyyyy and wwwwww are relocatable addresses
(dddd is FORTRAN source line count)

```
    ERROR SUMMARY

        ERROR        TIMES
        xxxxx        yyyy
                .
                .
                .
```

(all numbers are decimal)

NO OUTPUT FILE FOUND

Functions of Entry Points

Q8NTRY   Initialize I/O buffer parameters

STOP   Enter STOP in dayfile and begin END processing

EXIT   Enter EXIT in dayfile and begin END processing

END   Terminate all output buffers, print an error summary, transfer control to the main overlay if within an overlay; in any other case exit to monitor

SYSTEM=   Handles error tracing, diagnostic printing, termination of output buffers and either transfers to specified nonstandard error recovery addresses, terminates the job or returns to calling routine depending on type of error

SYSTEMC   Changes entry to SYSTEM's error table according to arguments passed

ABNORM=   Gains control from an execution routine when an error had been assembled as fatal and during the processing of the job was changed to nonfatal with no nonstandard error recovery. An abnormal termination is given.

FILE NAME HANDLING BY SYSTEM

SYSTEM(Q8NTRY) places in RA+2 and the locations immediately following, the file names from the FORTRAN PROGRAM card. The file name is left justified, and the file's FIT address is right justified in the word. (Thus the declared names replace any actual file names at execution time in the RA area.) The file name occupies 42 bits. The FIT address occupies 18 bits.

The logical file name (LFN) which appears in the first word of the FIT is determined in one of the three following ways:

CASE 1: If no actual parameters are specified, the LFN will be the file name from the PROGRAM card.

```
Example:  .
          .
          .
          RUN(S)
          LGO.
          .
          .
          .
          PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)

Before    SYSTEM(Q8NTRY)  is executed

RA+2      000            000
          000            000

After     SYSTEM (Q8NTRY)                  LFN in FIT

RA+2      INPUT          FIT address       INPUT
          OUTPUT         FIT address       OUTPUT
          TAPE1          FIT address       TAPE1
          TAPE2          FIT address       TAPE2
```

CASE 2: If actual parameters are specified, the LFN will be that specified by the corresponding actual parameter, or the file name from the PROGRAM card if no actual parameter was specified. A one-to-one correspondence exists between the actual parameters and the file names found on the PROGRAM card.

```
Example:  .
          .
          .
          RUN(S)
          LGO(,,DATA,ANSW)
          .
          .
          .
          PROGRAM TEST2(INPUT,OUTPUT,TAPE1,TAPE2,TAPE3=TAPE1)
```

Before	SYSTEM (Q8NTRY) is executed	
RA+2	000	000
	000	000
	DATA	000
	ANSW	000

After	SYSTEM (Q8NTRY)		LFN in FIT
RA+2	INPUT	FIT address	INPUT
	OUTPUT	FIT address	OUTPUT
	TAPE1	FIT address	DATA
	TAPE2	FIT address	ANSW
	TAPE3	FIT address of TAPE1	Uses TAPE1 FIT

CASE 3: An equivalenced file name from the PROGRAM card will ignore an actual parameter. The LFN will be that of the file to the right of the equivalence and no new FIT will be created.

Example:

```
.
.
.
RUN(S)
LGO(,,DATA,ANSW)
.
.
.
PROGRAM TEST3(INPUT,OUTPUT,TAPE1=OUTPUT,TAPE2,TAPE3)
```

Before	SYSTEM (Q8NTRY)	
RA+2	000	000
	000	000
	DATA	000
	ANSW	000

After	SYSTEM (Q8NTRY)		LFN in FIT
RA+2	INPUT	FIT address	INPUT
	OUTPUT	FIT address	OUTPUT
	TAPE1	FIT address of OUTPUT	Uses OUTPUT FIT
	TAPE2	FIT address	ANSW
	TAPE3	FIT address	TAPE3

FORTRAN CROSS REFERENCE MAP

If the ninth field of the run control card is non-zero, FORTRAN supplies the programmer with a cross-reference map after each PROGRAM, SUBROUTINE, or FUNCTION, purely as an aid to program debugging. The following information is furnished:

Program length including I/O buffers

Statement function references with the relative core locations, general compiler tag assigned, symbolic tag given in the program and the references to the statement function

Statement number references with the same information as above

Block names and lengths

Variable references - also with location, general tag, symbolic tag, and a list of references

Start of constants (relative address)

Start of temporaries (relative address)

Start of indirects (relative address)

Unused compiler space

The programmer should bear in mind that because of the operation of the compiler not all references will be listed. An actual physical reference is necessary before the reference is placed in the reference map. If the required variable address is already in a register, the compiler will use the address in the register and not make an actual variable reference by name. A reference to a statement number will not be listed if an actual jump is not necessary, such as when the code simply falls through to the next statement and the compilation of a jump instruction is therefore unnecessary.

FORTRAN II FEATURES

The following FORTRAN II statements are accepted by FORTRAN:

1. In FORTRAN II arithmetic replacement statements, column 1 may contain either of the following characters:

D	Double Precision mode
I	Complex mode

   When these characters are encountered, all variables and constants in the statement are assumed to be of the same type (double precision or complex).

2. FORTRAN II statements which contain a B in column 1 (Boolean) are evaluated as masking expressions. The operator equivalences are:

FORTRAN	FORTRAN II
.AND.	*
.NOT.	-
.OR.	+
none	/

   Exclusive OR function is defined as:

p	v	p/v
1	1	0
1	0	1
0	1	1
0	0	0

3. Mixed mode variables may appear in any FORTRAN II Boolean B-type statements.

4. Sense Light Statements

   SENSE LIGHT i

   The statement turns on SENSE LIGHT i; i must be an integer constant in the range 1 to 6.

   SENSE LIGHT 0 turns off all sense lights.

   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. The n_i are statement labels; i must be an integer constant in the range 1 to 6.

5. IF SENSE SWITCH Statement

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 constant (1 to 6).

6. Fault Condition Statements

At execute time, the computer may be set to interrupt on divide overflow or exponent fault. This is the default setting on SCOPE 2. The fault indicator may be checked immediately after any statement that could possibly cause a fault condition if the SCOPE 2 mode control card is used to inhibit the concerned interrupts.

IF DIVIDE CHECK n_1, n_2

A divide check occurs following division by zero. The statement checks for this condition. If it has occurred, a jump to statement n_1 takes place. If no check exists, a jump to statement n_2 takes place.

IF QUOTIENT OVERFLOW n_1, n_2

IF ACCUMULATOR OVERFLOW n_1, n_2

An overflow occurs when the result of a real, double precision, or complex arithmetic operation exceeds the upper limits specified for these types. Results that are less than the lower limits are get to zero without indication. This statement is therefore a test for floating point overflow only. If the condition has occurred, a jump to statement n_1 takes place. If the condition does not exist, a jump to statement n_2 takes place.

7. FORTRAN accepts the FORTRAN II version of the EXTERNAL statement. This form contains the same name list, but the word EXTERNAL has been replaced by the character F in column 1 of the statement.

```
F         name1,name2,....
```

8. The only inherently incompatible areas are the following:

COMMON-EQUIVALENCE Statement Relationships

In FORTRAN II, equivalence groups can reorder the common variables and arrays, and more than one variable in an equivalence group can be in common.

In FORTRAN, equivalence groups do not reorder common, but may only extend the length of a common block.

Function-Naming Conventions

In FORTRAN II, the following rules apply for function subprograms, library function and statement function names:

The name is 4-7 alphanumeric characters, ending with the character F.

The first character must be X if, and only if, the value of the function is integer; for any other first character, the value of the function is real.

In FORTRAN, the number of characters in the function name is 1-7; the first character must be alphabetic.

FORTRAN LIBRARY ROUTINE ENTRY POINTS

Routine	Entry Points	Externals
ACGOER	ACGOER	SYSTEM, ABNORMAL
ALNLOG	ALOG, ALOG10	SYSTEM
ASINCOS	ASIN, ACOS	SYSTEM
ATAN	ATAN	SYSTEM
ATAN2	ATAN2	SYSTEM
BACKSP	BACKSP	SYSTEM, ABNORML, GETBA, CIO1,. BKSPRU, FIZBAK
BUFFEI	BUFFEI	SYSTEM,ABNORML, GETBA, OPEN., CIO1.
BUFFEO	BUFFEO	SYSTEM, ABNORML, GETBA, OPEN., CIO1.
CABS	CABS	SYSTEM
CBAIEX	CBAIEX	SYSTEM
CCOS	CCOS	COS, SIN, EXP, SYSTEM
CEXP	CEXP	COS, SIN, EXP, SYSTEM
CLOG	CLOG	ALOG, ATAN2, CABS, SYSTEM
CSIN	CSIN	COS, SIN, EXP, SYSTEM
CSQRT	CSQRT	CABS, SQRT, SYSTEM
DABS	DABS	SYSTEM
DATAN	DATAN, DATAN2	SYSTEM
DBADEX	DBADEX, DBAREX, RBADEX	DLOG, DEXP, SYSTEM
DBAIEX	DBAIEX	SYSTEM
DBLE	DBLE	
DEXP	DEXP	SYSTEM
DISPLA	DISPLA	
DLNLOG	DLOG, DLOG10	SYSTEM
DMOD	DMOD	SYSTEM
DSIGN	DSIGN	SYSTEM
DSINCOS	DSIN, DCOS	SYSTEM
DSQRT	DSQRT	SYSTEM
DUMP	DUMP, PDUMP	OUTPUTC, STOP

FORTRAN LIBRARY ROUTINE ENTRY POINTS (Cont'd)

Routine	Entry Points	Externals
DVCHK	DVCHK	
ENDFIL	ENDFIL	SYSTEM, ABNORML, GETBA, FIZBAK., OPEN., CIOM.
EXP	EXP	SYSTEM
GETBA	GETBA	SYSTEM, ABNORML
IBAIEX	IBAIEX	SYSTEM
IDINT	IDINT	SYSTEM
IFENDF	IFENDF	SYSTEM, ABNORML, GETBA
INPUTB	INPUTB	SYSTEM, ABNORML, GETBA OPEN., CIO1., RDWDS.
INPUTC	INPUTC	SYSTEM, ABNORML, GETBA, KRAKER, OPEN., RDCARD., DAT.
INPUTS	INPUTS	SYSTEM, ABNORML, KRAKER
KODER	KODER	SYSTEM, ABNORML
KRAKER	KRAKER	SYSTEM, ABNORML
LEGVAR	LEGVAR	
LENGTH	LENGTH	SYSTEM, ABNORML, GETBA
LOCF	LOCF, XLOCF	
OUTPTB	OUTPTB	SYSTEM, ABNORML, GETBA OPEN., WRWDS., CIO1.
OUTPTC	OUTPTC	SYSTEM, ABNORML, GETBA, KODER, OPEN., WRWDS., DAT., FIZBAK.
OUTPTS	OUTPTS	SYSTEM, ABNORML, KODER
OVERFL	OVERFL	
OVERLAY	OVERLAY	LOADER, SYSTEM, ABNORML
PAUSE	PAUSE	
RANF	RANF	
RBAIEX	RBAIEX	SYSTEM
RBAREX	RBAREX	ALOG, EXP, SYSTEM
REMARK	REMARK	
REWINM	REWINM	SYSTEM, ABNORML, GETBA, CIO1.
SECOND	SECOND	

FORTRAN LIBRARY ROUTINE ENTRY POINTS (Cont'd)

Routine	Entry Points	Externals
SIO$	SIO., CIO1., OPEN., BKSPRU., FIZBAK	
SINCOS	SIN, COS	SYSTEM
SLITE	SLITE	SYSTEM
SLITET	SLITET	SYSTEM
SNGL	SNGL	
SQRT	SQRT	SYSTEM
SSWTCH	SSWTCH	SYSTEM
SYSTEM	SYSTEM, SYSTEMC, SYSTEMP, Q8NTRY, STOP, END, EXIT, ABNORML	
TAN	TAN	SYSTEM
TANH	TANH	EXP.SYSTEM
TIME	TIME	

LARGE CORE MEMORY INPUT/OUTPUT K

The following statements result in data transmission between LCM and SCM.

CALL READEC (a,b,n)

CALL WRITEC (a,b,n)

a	Simple subscripted variable located in SCM
b	Simple or subscripted variable located in LCM common block
n	Integer constant or integer expression

When either statement is executed, n consecutive words of data are transmitted between SCM and LCM beginning at location a in SCM and b in LCM. CALL READEC transfers words from LCM to SCM and CALL WRITEC transfers words from SCM to LCM.

OBJECT—TIME INPUT/OUTPUT L

STRUCTURE OF SEQUENTIAL I/O FILES

DEFINITIONS

Logical Record — An amount of data created/processed by any of the following:

- One unformatted WRITE/READ
- One BUFFER OUT/BUFFER IN
- One WRITMS/READMS
- One unit record (print line/punched card) defined within a formatted WRITE/READ

Physical Record (Block) — The data recorded between two interrecord gaps on a magnetic tape. Block structure may be simulated on mass storage files if specified by the user.

Definitions of file, logical file, and further information on logical/physical records is included in the CYBER 70/Model 76 SCOPE 2 Reference Manual.

Table L-1 provides brief descriptions of the block/record formats supported by CYBER 70/Model 76 SCOPE 2.

TABLE L-1. SCOPE 2 BLOCK/RECORD FORMATS

Record Type	Description
F	Record length is fixed
D	Record length given as character count in decimal by length field contained within record
B	Record length given as character count in binary by length field in first four characters of record
R	Record terminated by record mark specified by user
T	Record consists of a fixed length header followed by a variable number of fixed length trailers. Header contains trailer count field in decimal
U	Record length defined by user
W	Record length contained in a control word prefixed to record by operating system
Z	Record terminated by a 12-bit zero byte in the low order byte position of a 60-bit word
S	Record consists of zero or more blocks of a fixed size followed by a terminating block of less than the fixed size. An 8 character level number is appended to the last block

Block Types	Description
K	All blocks except possibly the last contain a fixed number of records
C	All blocks contain less than or equal to a fixed number of characters
E	All blocks contain an integral number of records. The block sizes may vary up to a fixed maximum number of characters
I	All blocks contain an internal control word prefixed to the block by the operating control word system

Table L-2 specifies those combinations of block record formats which can be processed by a FORTRAN program.

TABLE L-2. FORTRAN BLOCK/RECORD FORMATS

Record Type Block Type	F	D	R	T	U	W	Z	S	X
K	X	X	X	X	X	X	X		
C	X	X	X	X		X	X	X	X
E	X	X	X	X		X	X		
I						X			
Unblocked	X	X	X	X		X	X		

☒ = legal   □ = illegal   † Read only

FORTRAN Default Conventions (Sequential Files)

The FORTRAN compiler will automatically provide the following file attributes:

File organization = Sequential

Block type = No blocking unless STAGE or REQUEST tape is encountered.
I for W; C for S,X; K for other

Record type = W

External character code = Display code

Label type = Unlabeled

Maximum block length = 5120 characters

Positioning before first access = No rewind

Positioning of current volume before switch = Unload

Positioning after last access = No rewind

Processing direction = I/O

Error options = T (terminate) for READ/WRITE; AD (accept and display) for BUFFER I/O

Suppress multiple buffer = NO (Record manager anticipates user requirements)

A unit record is one W-format record. One formatted WRITE can create several unit records. One formatted READ can process as input several unit records.

A logical record is one W-format record. One unformatted WRITE or BUFFER OUT, or call on WRITMS creates one logical record. One unformatted READ or BUFFER IN or call on READMS processes as input one logical record.

Access to Additional Block/Record Types

The Record Manager FILE statement can be used by the FORTRAN programmer to override the default values supplied by the FORTRAN compiler in order to access record types other than W. The FILE statement is described in detail in the CDC CYBER 70/Model 76 SCOPE 2 Reference Manual. Only those block/record combinations that are legal in CDC CYBER 70/Model 76 SCOPE 2 can be used. Blocking may also be specified via a FILE card.

The FORTRAN language does not contain any statements that specify or constrain the block type. Therefore, the FILE card can specify any block type which is consistent with the record type.

Although the FORTRAN language does not contain statements that specify the record type, constraints are imposed on the processing of certain types of records due to the logical structure of the records themselves. The following table provides more detailed information on these constraints.

TABLE L-3. FORTRAN CONSTRAINTS

Record Type	User Action Required for Writing
D	User must insert the record length appropriately before write. The offset from the beginning of the record cannot be such that the length field lies outside the buffer utilized in the FORTRAN object time I/O routines
R	User must insert the record mark appropriately before write
T	User must insert the trailer count appropriately on write. The trailer count field must obey the same restrictions regarding the length of the FORTRAN object time I/O routine buffer as described for the record length in D format records
F	User must ensure that all records written are fixed length
Z	User must ensure that recording mode is set to binary for tape. System will not convert zero bytes to special codes but will pass them through to the device driver
U	User must ensure that only one record per block is written (that is, only block type K, one record per block , is allowed)
S	User must be aware that each unit record/logical record will be an S record

Example of FILE Statement Usage

The job deck pictured causes the FORTRAN default file attributes to be overridden with the following:

Label type - unlabeled

Block type - character count (C)

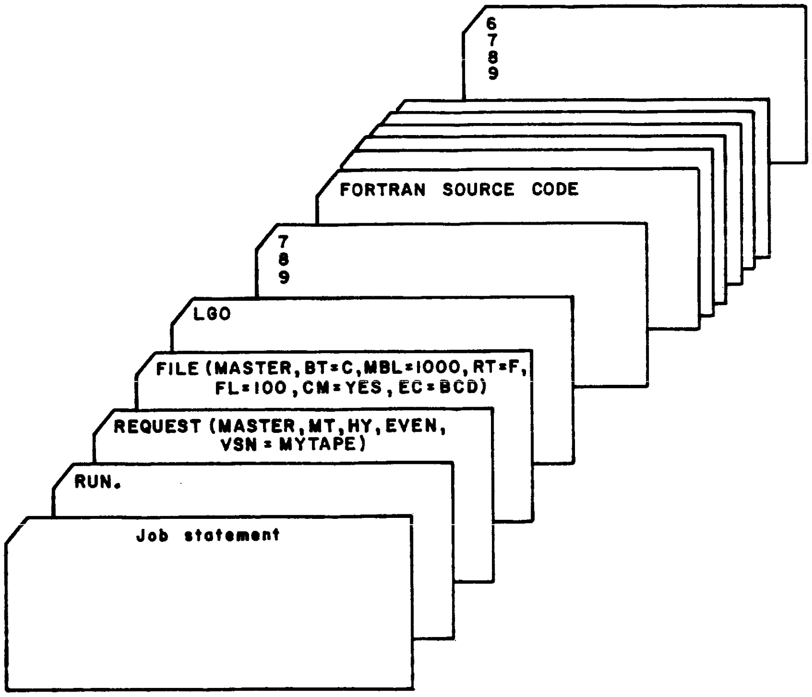
Maximum block length - 1000 characters

Record type - fixed length (F)

Record length - 100 characters

Conversion mode - yes

External code - BCD


6
7
8
9
FORTRAN SOURCE CODE
7
8
9
LGO
FILE (MASTER,BT=C,MBL=1000,RT=F, FL=100,CM=YES,EC=BCD)
REQUEST (MASTER,MT,HY,EVEN, VSN = MYTAPE)
RUN.
Job statement

Assuming the job is creating the file with formatted WRITE statements such that 100 character records are always written, the file will be written on magnetic tape in 1000 character blocks (except perhaps the last block) with even parity at 800 bpi. No labels will be recorded. Records will be blocked 10 to a block. No information will be written other than that supplied by the user.

BUFFER IN/BUFFER OUT (Sequential Files)

BUFFER IN/BUFFER OUT can be used to achieve some degree of overlap between the user program and the I/O transfer with an external device (mass storage or tape). However, the memory area specified in the BUFFER I/O statement will not be utilized as the physical record (block) buffer. These buffers are maintained within an operating system buffer area in LCM. The execution of a BUFFER I/O, will therefore involve movement of a logical record between system buffers in LCM and the memory area specified in the BUFFER I/O statement. Correspondence between individual BUFFER I/O statements and physical records (blocks) on a device depends upon the block specification. For example, K blocking with a record count of 1 will ensure that each BUFFER I/O corresponds to a block.

BUFFEI (BUFFER IN)

One, and only one, logical record is read each time BUFFEI is called. If the record length specified by the call is longer than the record read, the excess locations in the user area are not changed by the read. If the record read is longer than the length specified by the call, the excess words in the record are passed over. The number of 60-bit words transmitted to the user area may be obtained by referencing LENGTH.

After using a BUFFER I/O statement on unit i and prior to a subsequent reference to unit i or to the information, the status of the BUFFER operation must be checked by a reference to the UNIT function. This will ensure that the data requested has actually been transferred and the buffer parameters for the file have been properly restored.

If an attempt is made to BUFFER IN past an end-of-partition without testing for the condition by referencing the UNIT function, BUFFEI will abort the program with the diagnostic:

```
*BUF  IN**ENDFILE filename
```

If the last operation on the file is a write operation, no data is available to read. If a read is attempted, BUFFEI aborts the program with the diagnostic:

```
*BUF  IN**LAST OP WRITE, filename
```

If the starting address for the block is greater than the terminal address, BUFFEI aborts the program with the diagnostic:

```
*BUF  IN**FWA.GT.LWA, filename
```

If an attempt is made to BUFFER IN from an undefined file (a file that has not been declared on the PROGRAM card), BUFFEI aborts the job and issue the diagnostic:

```
*BUF  IN**UNASSIGNED MEDIUM, filename
```

BUFFEO (BUFFER OUT)

One logical record is written each time the routine is called. The length of the record equals terminal address minus address + 1.

As with BUFFER IN, a BUFFER OUT operation must be followed by a reference to the UNIT function.

If the starting address for the block is greater than the terminal address, BUFFEO aborts the program with the diagnostic:

```
*BUF  OUT**FWA.GT.LWA, filename
```

CONFLICT diagnostic is similar to that issued by BUFFEI.

The UNASSIGNED MEDIUM diagnostic is similar to that issued by BUFFEI.

Random Access Files

Random access I/O operations are implemented by using the word addressable file capability in CYBER 70/Model 76 SCOPE 2.

Two degrees of sophistication are available using the mass storage subroutines. It is possible to use the routines in a normal way having just one master index, or it is possible to have a master index and many subindexes. A file may have a name or a number index.

In all cases it is necessary to open (CALL OPENMS) the mass storage file before calling READMS, WRITMS, or STINDX. If the file exists, OPENMS reads the master index into the area specified in the call (the ix parameter).

The STINDX subroutine causes no transfer of data, it merely changes the file index to the base specified in the call. A new subindex is created by a call on STINDX to change the file index base followed by calls on WRITMS to create the new subindex and transfer the records indexed by the new subindex. STINDX must then be called again to reset the file index base to the previous level index followed by a call on WRITMS to transfer the new subindex to mass storage.

A new subindex is read in by calling READMS to read in the subindex followed by a call on STINDX to change the file index base to the new subindex.

After making a call to STINDX, if the next operation of file u is to be a random access write (WRITMS) and if the file is being referenced through a name index, the programmer must zero out the area reserved for the new index buffer (whose first address is specified by the ix parameter in the call to STINDX) prior to calling WRITMS. The master index must be reset by a call on STINDX before termination of the job so that the correct index will be written on the file.

Upon termination of the job the file is closed automatically by FORTRAN. At this time the index is dumped to the file.

Example 1.

```
            PROGRAM MS (TAPE5)
            DIMENSION I (10), B(20), C(30)
            CALL OPENMS(5, I, 10, 0)
   C READ MASTER INDEX INTO I
            .
            .
            .
            CALL READMS (5, B, 20, 4)
   C READ RECORD 4 INTO B (ASSUME THIS RECORD IS A SUBINDEX)
            CALL STINDX (5, B, 20)
   C ALL SUBSEQUENT OPERATIONS ON UNIT 5 WILL USE
   C B AS THE INDEX FOR THE FILE
            .
            .
            .
            CALL STINDX (5, I, 10)
   C RESTORE MASTER INDEX
            END
```

Example 2.

```
            PROGRAM MS (TAPE5)
   C PROGRAM FOR CREATING RANDOM FILE
            DIMENSION J(10), B(5), XYZ(20), ZXY(10), YXA(50)
            CALL OPENMS(5, J, 10, 1)
            CALL STINDX(5, B, 5)
   C USE INDEX B
            CALL WRITMS(5, XYZ, 20, JOE)
            CALL WRITMS(5, ZXY, 10, SAM)
            CALL WRITMS(5, YXA, 50, PETE)
            CALL STINDX(5, J, 10)
            CALL WRITMS(5, B, 5, SUB1)
   C WRITE OUT THE SUBINDEX
            END
```

Example 3.

```
          PROGRAM MS (TAPE5)
   C THIS MS FILE HAS NO SUBINDEXES
          DIMENSION I(10)
          CALL OPENMS(5, I, 10, 0)
   C READ MASTER INDEX INTO I
          .
          .
          .
          .
          .
   C ANY READ OR WRITE ON THIS FILE WILL USE THE INDEX IN C ARRAY I
          .
          .
          .
          .
          .
          END
```

FORTRAN Default Conventions (Random Files)

The FORTRAN compiler will automatically provide the following file attributes when a file is processed using the mass storage subroutines:

File organization = Word addressable (wa)

Block type = Not applicable

Record type = W

External character code = Not applicable

Label type = Unlabeled

Maximum block length = Not applicable

Positioning before first access = Rewind

Positioning of current volume before switch = Not applicable

Positioning after last access = Rewind

Processing direction = I/O

Error options = T (terminate)

Suppress multiple buffer = YES (Record manager does not anticipate file accesses)

Conversion mode = Not applicable

One WRITMS operation creates one W-format record. One READMS operation processes as input one W-format record. The master index is the first W-format record in the file. Refer to Random Access Files description preceding for user responsibilities. If the length specified by a call on READMS is longer than the record, the excess locations in the user area are not changed by the read. If the record is longer than the length specified on a call on READMS, the excess words in the record are passed over.

Status Checking

UNIT (I) Function

The UNIT (I) function is used to check the status of the last previous buffered operation, (BUFFER IN or BUFFER OUT only) on logical unit i. The function returns values as follows:

< 0 - Unit busy

=0 - Unit ready

>0 - End-of-partition encountered on previous read

> 1 - Parity error

Example:

```
IF(UNIT,I) 12,14,16,18
```

Upon return from the UNIT function, control is transferred to the statement labeled 12, 14, or 16 if the value returned was < 0, 0, or > 0 respectively.

If the value returned is < 0 or > 0 the condition indicator is cleared before returning to program control.

NOTE

> If the UNIT function references a non-buffered unit (a unit referenced by I/O statements other than BUFFER IN and BUFFER OUT), the status returned will always indicate unit ready and no previous error (-1).

If any of the following conditions exist following the previous read, an end-of-file status will be returned:

Deleted W-format flag record encountered (INPUT file only ignored on other files This appears for a 7/8/9 end-of-section card).

End of information encountered

Non-deleted W-format flag record encountered

Embedded tape mark encountered (EOP)

Terminating double tape mark encountered

Terminating EOF label encountered

Embedded zero length level 17 block encountered

EOF Function

The EOF (i) function is used to test for end-of-partition (non buffered) on unit i. The value zero is returned if no end-of-partition was encountered on the previous read, or non-zero if end-of-partition was encountered on· unit i.

Example:

```
    IF(EOF,I) 10,20
```

Upon return, control is transferred to the statement labeled 10 if the previous read encountered an end-of-partition, or to 20 if not.

If an end-of-partition is encountered, EOF clears the indicator before returning control.

NOTE

> The user should make the EOF check after each READ operation to ensure against possible I/O errors. If a READ on unit i is attempted and an EOP was encountered on the previous READ operation, execution is terminated and a diagnostic message issued.
>
> If the previous operation on unit i was a write, the test always returns a zero value. Only when an end-of-partition was read will the end-of-partition condition exist.
>
> This function has no meaning when applied to a word addressable file. If the EOF function is called in reference to a word addressable file, a zero value is always returned.

If any of the following conditions exist following the previous read, an end-of-partition is returned.

Deleted W-format flag record encountered (INPUT file only, ignored on other files. This appears for a 7/8/9 end-of-section card).

End of information encountered

Non-deleted W-format flag record encountered (EOP for W records)

Embedded tape mark encountered (EOP)

Terminating double tape mark encountered (EOI)

Terminating EOF label encountered (EOI)

Embedded zero length level 17 block encountered (EOP for S and Z records types with C blocking, only)

IOCHEC Function

The IOCHEC (i) function is used to test for parity errors on non-buffered reads on unit i. The value zero is returned if no error occurred.

Example:

```
     J     IOCHEC (i)
     IF    (J) 15,25
```

A value of zero is returned to J if no parity error has occurred. Otherwise, non-zero is returned. Control then transfers to the statement labeled 25 or 15, respectively.

If a parity error has occurred, IOCHEC clears the parity indicator before returning.

Parity errors are handled in the above fashion regardless of the type of the external device.

Parity Error Detection

Parity errors are detected by the status checking functions on all read operations and on write operations that access mass storage files for which the write check option has been included on the REQUEST statement for the file. The REQUEST statement is described in the CYBER 70/Model 76 SCOPE 2 Reference Manual.

Write parity errors for other types of devices (staged/on-line tape) are detected by the operating system and a message written in the dayfile.

When a parity error status is returned, it does not necessarily refer to the immediately preceding operation because of record blocking/deblocking performed by the data manager I/O routine via buffers in LCM.

BACKSPACE/REWIND

A BACKSPACE operation will always cause the referenced file to be logically moved back one record. A physical repositioning of the file on the external I/O device may be required. As a result of a BACKSPACE the last record becomes the next record.

BACKSPACE is permitted only for files with F, S, or W record format or tape files with one record per block.

The user should remember that formatted I/O operations can read/write more than one record whereas unformatted I/O and BUFFER IN/BUFFER OUT read/write only one record.

The REWIND operation positions a magnetic tape file such that the next FORTRAN I/O operation will reference the first record. A mass storage file will be positioned to the beginning of information.

Table L-4 provides greater detail as to the actions performed prior to positioning for various conditions.

TABLE L-4. BACKSPACE/REWIND OPERATION

BACKSPACE/REWIND		
Condition	Device Type	Explanation
Last operation was WRITE or BUFFER OUT ENDFILE	Mass Storage (no blocking)	1. Any unwritten blocks for the file are written 2. If record format is S a zero length level 17 block is written
	Unlabeled Magnetic Tape or Blocked Mass Storage	1. Any unwritten blocks for the file are written 2. If record format is S a zero length level 17 block is written 3. Two file marks are written
	Labeled Magnetic Tape or Labeled Blocked Mass Storage	1. Any unwritten blocks for the file are written 2. If record format is S a zero length level 17 block is written 3. A file mark is written 4. A single EOF1 label is written 5. Two file marks are written
Last operation was READ, BUFFER IN or BACKSPACE	Mass Storage	None
	Unlabeled Magnetic Tape	None
	Labeled Magnetic Tape	If the end of information has been reached then labels are processed
No previous operation	Magnetic Tape	1. If the file is assigned to on-line magnetic tape a REWIND request will be executed 2. If the file is staged the REWIND request has no effect. The file will be staged and rewound when first referenced
	Mass Storage	The REWIND request will cause the file to be rewound when first referenced
Previous operation was REWIND		Current REWIND is ignored

ENDFILE

The ENDFILE operation introduces a delimiter into an I/O file. The following table provides detailed information as to the effect of ENDFILE for various record types.

TABLE L-5. ENDFILE OPERATIONAL EFFECTS

	ENDFILE
Record Type	Action
W	Write non-deleted W-flag record Terminate current block if magnetic tape file
S	Write level 17 zero length block
Z with C blocking	Write level 17 zero length block
D, B, R, T, F, U, or Z	Terminate current block if magnetic tape file and write tape mask. No other data is written

NOTE

> A WRITE/BUFFER OUT can legally follow an ENDFILE operation. If the file has records of the format W, S, or Z with C blocking or is a mass storage file with any other block/record formats no special action is performed. However, if the file is assigned to magnetic tape and has a record format other than W, S, or Z with C blocking, a tape mark will be written preceding the next block record.

FORMAT/ENCODE/DECODE

FORMAT Field Separators

Field descriptors are normally delimited by field separators; however, some permissiveness is allowed.

Example:

```
        10 FORMAT (F25.22F10.3)
```

would be interpreted as two descriptors, F25.22 and F10.3, but perhaps the programmer meant F25.2 and 2F10.3. If there could be any ambiguity, always use field separators.

ENCODE/DECODE

Under 64/6600 SCOPE a binary zero byte is used to terminate a unit record. Whenever the DECODE processor encounters a zero character (6 bits of binary zeros), that character will be interpreted as a blank. Conversion will continue through n characters per record.

Whenever a record terminator (a / or the final, if the list is not exhausted) is encountered in a FORMAT statement, the rest of the record will be filled out with blanks (for ENCODE) or ignored (for DECODE), and conversion will continue beginning with the next record. [The length of record is specified by n in a DECODE (n, f, A) or ENCODE (n, f, A) k statement.] The record is restricted to a maximum length of 150 characters.

Example:

> 10 FORMAT (16(F10.4)) is illegal (the diagnostic EXCEEDED RECORD SIZE is issued in this case)
>
> 10 FORMAT (10F10.4/6F10.4) is allowed

Formatted Output of Out of Range Data

A FORTRAN formatted WRITE will produce X's or I's in an output field under the following conditions:

1. Fixed point format will produce X s in the output field if the internal data is out of range (greater than or equal to 2**48).
2. Floating point format will produce X's in the output field if the internal data is out of range or I's if it is indefinite (as defined for CYBER 70/Model 76 Hardware.)

Restrictions

Meaningful results are not guaranteed in the following circumstances.

Mixing BUFFER I/O statements and standard READ/WRITE statements on the same file (without a REWIND in between).

Requesting a LENGTH function on a BUFFER unit before requesting a UNIT function.

Two consecutive BUFFER I/O statements on the same file without the intervening execution of a UNIT function call.

Standard Labeled Files

Only files recorded on magnetic tape can be labeled files. A LABEL statement (described in CYBER 70/Model 76 SCOPE 2 Reference Manual) is required to process labeled files through a FORTRAN program. The LABEL statement must at least specify values for the following parameters:

W	-	Create labels (indicates no labels exist or create new ones)
R	-	Check labels (indicates labels exist on tape)

This parameter is necessary because the processing direction for sequential files in FORTRAN must be input/output in order to permit both READ and WRITE by the FORTRAN program. Therefore it is necessary that the user specify whether a labeled magnetic tape file is utilized for input (R-check labels) or output or input/output (W-create labels).

Label information used for checking/creating labels can be presented to the operating system in either of the following two ways:

LABEL statement

LABEL subroutine - An object time subroutine is provided to allow the FORTRAN programmer to generate the information necessary for label checking/creation during execution. If the label information is properly set up and the subroutine LABEL is referenced prior to any other reference to the file, then when the first reference occurs the label and the information are compared for an input tape, or the information is written on an output tape.

The form of the call is:

```
CALL    LABEL  (u,fwa)
```

u is the unit number

fwa is the first word address of the label information. The label information must be formatted as described for the LABEL macro parameter list in the CYBER 70/Model 76 SCOPE 2 Reference Manual.

Automatic OPEN/CLOSE

The CYBER 70/Model 76 SCOPE 2 Reeord manager requires that an OPEN operation be performed before a file is referenced and a CLOSE operation be performed before job step termination. Since the FORTRAN language does not explicitly contain these functions, they must be performed implicitly by the FORTRAN object time I/O routines.

OPEN

The first reference to a file, or any reference to a closed file by a READ, BUFFER IN, WRITE, or BUFFER OUT statement, causes the FORTRAN object time I/O routines to perform an OPEN function for the file. If the file is labeled, header label processing occurs.

CLOSE

The FORTRAN object time I/O routines performs a CLOSE operation for a file which is open when the job step terminates. With the exception of positioning, the effects of a CLOSE operation are identical to those described for REWIND in the following cases:

Last operation was WRITE/BUFFER OUT
Last operation was READ/BUFFER IN/BACKSPACE
Last operation was ENDFILE

A CLOSE operation leaves a magnetic tape file positioned between the two terminating tape marks. A mass storage file is not repositioned in any way.

A REWIND operation performed as a FORTRAN statement is effectively a CLOSE with positioning.

FILE NAME HANDLING

FORSYS=(Q8NTRY) places in RA+2 and the locations immediately following, the file names from the FORTRAN PROGRAMS card. The file name is left justified, and the file's FIT address is right justified in the word. (Thus the declared names replace any actual file names at execution time in the RA area.) The file name occupies 42 bits. The FIT address occupies 18 bits.

The logical file name (LFN) which appears in the first word of the FIT is determined in one of the three following ways:

CASE 1: If no actual parameters are specified, the LFN will be the file name from the PROGRAM card.

```
Example:  .
          .
          .
          RUN(S)
          LGO.
          .
          .
          .
          PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)

Before    FORSYS=(Q8NTRY) is executed

RA+2      000            000
          000            000

After     FORSYS=(Q8NTRY)               LFN in FIT

RA+2      INPUT          FIT address    INPUT
          OUTPUT         FIT address    OUTPUT
          TAPE1          FIT address    TAPE1
          TAPE2          FIT address    TAPE2
```

CASE 2: If actual parameters are specified, the LFN will be that specified by the corresponding actual parameter, or the file name from the PROGRAM card if no actual parameter was specified. A one-to-one correspondence exists between the actual parameters and the file names found on the PROGRAM card.

```
Example:  .
          .
          .
          RUN(S)
          LGO(,,DATA,ANSW)
          .
          .
          .
          PROGRAM TEST2(INPUT,OUTPUT,TAPE1,TAPE2,TAPE3=TAPE1)
```

Before	FORSYS=(Q8NTRY) is executed	
RA+2	000	000
	000	000
	DATA	000
	ANSW	000

After	FORSYS=(Q8NTRY)		LFN in FIT
RA+2	INPUT	FIT address	INPUT
	OUTPUT	FIT address	OUTPUT
	TAPE1	FIT address	DATA
	TAPE2	FIT address	ANSW
	TAPE3	FIT address of TAPE1	Uses TAPE1 FIT

CASE 3: An equivalenced file name from the PROGRAM card will ignore an actual parameter. The LFN will be that of the file to the right of the equivalence and no new FIT will be created.

Example:

```
.
.
.
RUN(S)
LGO(,,DATA,ANSW)
.
.
.
PROGRAM TEST3(INPUT,OUTPUT,TAPE1=OUTPUT,TAPE2,TAPE3)
```

Before	FORSYS=(Q8NTRY)	
RA+2	000	000
	000	000
	DATA	000
	ANSW	000

After	FORSYS=(Q8NTRY)		LFN in FIT
RA+2	INPUT	FIT address	INPUT
	OUTPUT	FIT address	OUTPUT
	TAPE1	FIT address of OUTPUT	Uses OUTPUT FIT
	TAPE2	FIT address	ANSW
	TAPE3	FIT address	TAPE3

Case 4: Line limit may be altered at execution time as an added parameter on the EXECUTE or load-and-execute control card.

Example: LGO (LC=2000)

The LC parameter may appear anywhere in the parameter list of the execution control card. It is not counted as a file name for file equivalencing purposes. The numeric value is interpreted as an octal number.

INDEX

COMMENT SHEET

MANUAL TITLE CONTROL DATA® CYBER 70/MODEL 76 COMPUTER SYSTEM/ 7600 COMPUTER SYSTEM FORTRAN RUN, Version 2 Reference Manual

PUBLICATION NO. 60360700 REVISION C

FROM: NAME: ______________________________

BUSINESS ADDRESS: ______________________________

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN USA

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Technical Publications Dept.
4201 North Lexington Avenue
St. Paul, Minnesota 55112

ARH220

CUT ALONG LINE

FOLD

FOLD

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 5544
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WOR

LITHO