CONTROL DATA
CORPORATION

CONTROL DATA®
6000 COMPUTER SYSTEMS

FORTRAN EXTENDED REFERENCE MANUAL
6000 VERSION 3

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original publication.
B	Revised December 1967.
C	Project updating of system and corrections in response to user comments.
(1-6-69)	
D	This revision includes changes required by release of SCOPE 3.1.6, and minor
(12-12-69)	corrections to the text in response to user comments or errata. Pages affected
	are: iii thru viii; 1-1, 2, 5, 6, 8 thru 10; 2-9; 3-1; 4-5, 9, 12; 5-4, 6, 9; 6-1
	thru 3, 6-7 thru 10, 12, 6-16 thru 18; 7-2 thru 4; 8-1, 6, 8, 9, 10; 9-2, 9-2.1, 10;
	B-2, 7, 8; C-2; D-1, 2, 3, 5; E-1; F-1 thru 4; G-1, 3, 5, 6 thru 11; H-5; I-1
	thru 3, 7, 8; J-1, 2; Index 1 thru 9; Comment Sheet.
E	Information included in this revision reflects changes made for version 3.0
(7-23-70)	which runs under SCOPE version 3.2. Pages affected are: iii thru ix;
	1-1 thru 1-10; 5-1, 5-2, 5-11; 6-1 thru 6-3, 6-17 thru 6-19; 9-6; 10-3;
	11-1 thru 11-24; 12-1 thru 12-10; C-1 thru C-10; D-1, D-4; G-5, G-9;
	H-1; I-1, I-3, I-5, I-7; K-1; Index-1 thru Index-11; Comment Sheet.
F	Project updating of system and additional debugging information. Pages
(1-15-71)	affected are: viii, ix; 1-5; 4-3; 5-11; 6-17, 6-18; 9-6 thru 9-8; 11-1 thru 11-24;
	12-1 thru 12-4, 12-9 thru 12-11; C-1 thru C-4, C-9; D-1, D-2; I-4; Index-1
	thru Index-11; Comment Sheet.
Publication No. 60176600	

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Documentation Department
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

PREFACE

This publication describes the features of the FORTRAN Extended language (version 3.0) for the CONTROL DATA® 6400/6500/6600/6700 Computers. It is assumed that the reader has some knowledge of an existing FORTRAN language and the CONTROL DATA 6400/6500/6600/6700 Computer System. The language described herein is an extension of the ANSI FORTRAN language.

The FORTRAN compiler operates in conjunction with version 1.1 COMPASS assembly language processor under the control of the SCOPE operating system (version 3.2). The FORTRAN processor makes optimum use of storage both during compilation and in generated machine language instructions. Implementation of this processor provides the capability of compilation and execution within a single job operation as well as the simultaneous compilation of several programs, utilizing the system's multi-programming features.

CONTENTS

FORTRAN CODING FORM

PROGRAM SAMPLE PROGRAM **CONTROL DATA CORPORATION** **NAME**

ROUTINE **DATE** **PAGE** **OF**

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT (0 = ZERO, Ø = ALPHA O; 1 = ONE, I = ALPHA I; 2 = TWO, Z = ALPHA Z)	SERIAL NUMBER
C	PRØG		RAM I S THE S Ø L U TI Ø N Ø F A N M DEGREE PØLYNØMIAL BY SQUARE ROOTS	PØLY1
			REAL X(100),Y(100),W(211),Z(111),A(11),B(11,12)	PØLY2
	1		FØRMAT(I2,I3/4F14.7)	PØLY3
	2		FØRMAT(5E15.6)	PØLY4
			READ (5,1) M,N,(X(I),I=1,N)	PØLY5
			LW=2*M+1	PØLY6
			LB=M+2	PØLY7
			LZ=M+1	PØLY8
			DØ 5 J=2,LW	PØLY9
	5		W(J)=0.0	PØLY10
			W(1)=N	PØLY11
			DØ 6 J=1,LZ	PØLY12
	6		Z(J)=0.0	PØLY13
			DØ 16 I=1,N	PØLY14
			P=1.0	PØLY15
			Z(1)=Z(1)+Y(I)	PØLY16
			DØ 13 J=2,LZ	PØLY17
			P=X(I)*P	PØLY18
			W(J)=W(J)+P	PØLY19
	13		Z(J)=Z(J)+Y(I)*P	PØLY20

FORM 252-B

Figure 1-1

PROPERTIES AND ELEMENTS OF FORTRAN 1

1.1 THE FORTRAN CHARACTER SET

Alphabetic:	A to Z	
Numeric:	0 to 9	
Special:	= equals	) right parenthesis
	+ plus	, comma
	- minus	. decimal point
	* asterisk	$ dollar sign
	/ slash	space (i.e., blank)
	(left parenthesis	

In addition, any character of the SCOPE character set may be used in Hollerith information and in comments.

1.2 FORTRAN STATEMENTS

FORTRAN source programs consist of an ordered set of statements from which the compiler generates machine instructions and constants. These statements describe a procedure to be followed during execution of the program.

The statements comprising the FORTRAN program are written in the following columns:

	Column	Content
Statements	1-5	Statement label (optional)
	6	Blank or zero
	7-72	FORTRAN statement
	73-80	Identification field
Statement Continuations	1-5	Ignored
	6	FORTRAN character other than blank or zero
	7-72	Continued FORTRAN statement
	73-80	Identification field
Comments	1	C or $ or *
	2-80	Comments

Except in Hollerith constants, blanks may be used freely and are ignored by the compiler. A coding line may contain more than one FORTRAN statement if each statement is separated by the special character $. The next column following $ is interpreted the same as column 7 of a normal statement. A $ may serve as a statement separator for all statements except FORMAT, END, or labeled statements.

Continuation

Any FORTRAN statement except a comment, END statement, or loader directive may be continued. A statement may be continued on as many as 19 lines, each denoted by a continuation character (any acceptable character other than blank or zero) in column 6 on the continuation card. A blank or zero in column 6 denotes the first line of a statement. Blank cards within the input deck are ignored by the compiler; however, a continuation card following a blank card is treated as a new statement. (See also chapter 11, Debugging Facility.)

Comment

Comment information is designated by a C, *, or $ in column 1 of a statement. A comment statement has no effect upon the program. Comments may be used to explain the logic of the program. They appear on the listing 2 through 80. Comments may not be continued by use of a continuation character in column 6.

Statement Label

Statements are identified by unsigned integers which can be referred to from other sections of the program. A statement label (from 1-99999) may be placed anywhere in columns 1-5 of the initial line of a statement. Leading zeros are ignored. In any program unit, each statement label must be unique.

Identification Field

The FORTRAN Extended compiler is designed so that input lines may be greater than 80 characters long (e.g., when the input medium is a file produced by one of the source editing programs such as UPDATE). Only the first 72 characters are processed by the compiler and only the first 100 characters appear on the listing. Positions beyond 72 may be used for identification codes or sequencing.

1.3 SYMBOLIC NAMES

A symbolic name may be any alphabetic character followed by 0-6 alphanumeric characters. It may not contain special characters. Embedded blanks are ignored. Symbolic names are used for: subprogram and subroutine names, function names, variables, block data program, main program, input/output unit, common block, and namelist group names.

1.4 DATA TYPES

Each of the seven types of data has different significance. The types are: integer, real, double precision, complex, logical, octal, and Hollerith.

Integer type may assume only whole number values. For multiplication and division of integer operands, the result is truncated to 48 bits. For addition and subtraction, the full 60-bit word is used.

Real type data is carried in normalized floating point form. The magnitude of values of real type data is in the range 10^{322} to 10^{-293} with approximately 15 significant digits and 14 digit precision.

Double precision data is similar to type real, but it has approximately 29 significant digits.

Complex data consists of an ordered pair of real data. Each part has the same precision as real data. The first part is the real part, and the second is the imaginary part.

Logical data has only a true or false value. True is represented by any negative value, and false is represented by any positive value.

Octal data may consist of any value from 0-7...7 which can be represented in a maximum of 60 bits (20 octal digits).

Hollerith data consists of strings of characters. Blank characters are valid in a Hollerith string.

1.5 CONSTANTS

A constant is an unvarying quantity. The types of constants are the same as the types of data.

1.5.1 INTEGER

An integer constant is a string of up to 18 decimal digits with a magnitude no larger than $2^{59}-1$. If multiplication or division is specified, the operands and result will be truncated to 48 bits. Effectively, an integer constant string may contain up to 15 decimal digits with a maximum magnitude of $2^{48}-1$. It may not contain embedded commas. For example:

```
  0          -2145637
 67          45753576357
345          -77
```

The result of integer addition or subtraction must not exceed $2^{59}-1$. Integers used as subscripts and DO indexes are limited to $2^{17}-2$. The integer constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be within the allowed magnitude.

The maximum value of an integer constant as a result of a conversion from a real constant is $2^{48}-1$. The maximum value of an integer constant as a result of multiplication or division must not exceed $2^{48}-1$. If the value should exceed the magnitude allowed, the high order bits are lost.

1.5.2 REAL

A real constant may be represented by a string of up to 15 significant decimal digits. It contains a decimal point or an exponent representing a power of 10, or both. Real constants may be in the following forms:

n.n n. n.nE±s n.E±s nE±s .nE±s

n is the coefficient; E signifies that the succeeding datum is the exponent; and s is the base 10 exponent. The value of s must be in the range -308 to +337. The plus sign may be omitted if s is positive. The magnitude of non-zero absolute real values may be in the range 10^{-293} to 10^{322}, with up to 15 significant digits. If the range of the real constant is exceeded, the constant is considered zero and a compiler diagnostic is issued.

Examples:

3.E1 (means 3.0×10^{1}; or 30.)

```
3.1415768          31.41592E-01
314.07             .31415E01
-3.14159E+279      .31415E+01
30E02              -30E02
```

1.5.3 DOUBLE PRECISION

A double precision constant is written as a string of digits and represented internally by two words. The forms are:

.nD±s n.nD±s n.D±s nD±s

The D must always appear; the coefficient is n; s is the exponent of base 10.

The plus sign may be omitted for positive s. The range is the same as that of a real constant but is accurate to approximately 29 decimal digits. If the range is exceeded, a compiler diagnostic is issued.

Examples:

```
3.1415927D+1     3141.593D3
3.1416D0         31416.D-04
3131.593D-03     31416D02
```

1.5.4 COMPLEX

A complex constant is an ordered pair of signed or unsigned real constants, separated by a comma, and enclosed in parentheses (r1,r2). r1 represents the real part of the complex number; r2 represents the imaginary part. r1 and r2 must adhere to the magnitude specified for real constants. If this range is exceeded, a compiler diagnostic is provided. Diagnostics also occur when the pair contains integer constants, including (0,0).

Examples:

FORTRAN Representation	Complex Numbers
(1.,6.55)	1. + 6.55i
(15.,16.7)	15. + 16.7i
(-14.09,1.6E-03)	-14.09 + .0016i
(0.,-1.)	0. - 1.0i

1.5.5 LOGICAL

Logical constants assume only the values of true or false. When the compiler generates a value for the constant .TRUE., it will generate a minus one; for the constant FALSE., a zero is generated. Logical constants must be preceded and followed by a period and have the forms:

```
.TRUE.  or  .T.
.FALSE.  or  .F.
```

Example:

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

1.5.6 HOLLERITH

A Hollerith constant is of the form hHf, hRf (right justified), or hLf (left justified). h is an integer constant whose value is greater than zero; f represents the Hollerith data and must contain exactly h characters. When the hHf form is used, if h is not a multiple of 10, the last word is left justified and blank filled. Incomplete words in the hRf and hLf forms are binary zero filled.

Blanks are significant in a Hollerith data string. Hollerith constants are stored internally in display code. (See Appendix A.)

Hollerith constants may be used in arithmetic expressions, DATA and CALL statements, and in function argument lists. If the constant is an operand of an arithmetic operation, an informative diagnostic to that effect is issued.

Examples:

```
6HCOGITO
4LERGO
3RSUM
3HSUM
```

The maximum number of characters allowed in a Hollerith constant depends on its usage. In an expression, h may not be greater than 10; in a DATA statement, h is limited only by the number of characters that can be contained in a maximum of 19 continuation lines. If more than 10 characters are given in a DATA statement for such a constant, only the last word will have the appropriate fill.

1.5.7 OCTAL

An octal constant consists of 1 to 20 octal digits followed by a B. The form is:

$n_1..n_iB$

If the constant exceeds 20 digits, or if a non-octal digit appears, a fatal compiler diagnostic is issued. When fewer than 20 octal digits are specified, the digits are right justified and zero filled.

Example:

```
2374216B
7777776B
777000777000777B
```

1.6 VARIABLES

A variable is a symbolic representation of a quantity that may assume different values during execution of a program.

1.6.1 VARIABLE NAMES

A variable name may be any combination of 1 to 7 alphanumeric characters, must begin with an alphabetic character, and may contain embedded blanks. It may not contain special characters. For a main program, the program name may not appear as a symbolic name in any statement other than the PROGRAM statement.

1.6.2 TYPES OF VARIABLES

The type of a variable may be declared explicitly with the FORTRAN type declarations. (The type of the data is converted to the type of the variable.)

For example:

```
INTEGER ABC123, GNU12, CATXXX, FIREOUT, JOKER
REAL ISPY, JASONII, KOOR47, NVRT, SAMPLE
```

If integer and real variables are not declared explicitly, the type is determined by the first character of the symbolic name. If the name begins with I, J, K, L, M, or N, the variable is assumed to be integer.

I15, JK26, KKK, LB02, NP456L, and MM are classed as integer variables and must adhere to all limitations stated for that type. Variables beginning with characters A-H and O-Z are considered to be real and must adhere to all limitations stated for that type.

Complex, logical, and double precision variables must be declared explicitly by a type declaration. The values which the variables represent must adhere to the limitations stated for the corresponding type of constant.

Octal and Hollerith data can be entered into or used in any type variable. When an octal or Hollerith constant is used in an arithmetic operation, it is used as is without conversion. If the constant in question is not combined with another type of variable or constant, it is considered to be of integer type.

Examples:

```
JX = 7HACCOUNT
```
JX is an integer variable containing a Hollerith constant.

```
IITT = 357215B
```
IITT is an integer variable containing an octal constant.

BC = 174B + 623B

Addition of octal constants is treated as an addition of two integer constants; the result is converted to the type defined for BC and stored.

KLM = 3.14 - 35B

KLM is defined as integer. The octal constant assumes the type of the other operand (real) and the result is real. That result is converted to integer before being stored in KLM.

1.6.3 EXTENDED CORE STORAGE (ECS)

An ECS variable must be defined explicitly by a type declaration. This type of data occupies a 60-bit word and resides in Extended Core Storage. ECS variables may appear in the source program only in the following circumstances:

In a COMMON statement as an element of an ECS common block

In a CALL or function reference as an actual parameter

In a SUBROUTINE or FUNCTION statement as a dummy parameter

In a TYPE ECS statement

In a DIMENSION statement

Only one common block may contain ECS variables, and all variables in the block must be of type ECS.

1.6.4 ARRAYS

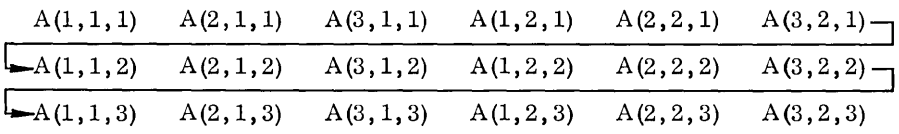
An array is an ordered set of variables identified by a variable name. Each variable in the array is referred to by the array name followed by a subscript which indicates its relative position within the array. The entire array may be referenced by the array name without subscripts when used as an item in an input/output list or in a DATA statement. In an EQUIVALENCE statement, however, only the first element of the array is implied by the unsubscripted array name.

Arrays may have one, two, or three dimensions and must be defined at the beginning of the program in a DIMENSION, COMMON, or type statement. When a reference is made to an array, if the subscripts exceed the magnitude of the dimensions declared initially, a position outside the array will be accessed. If the number of subscripts is greater than the number of dimensions defined, a diagnostic is issued.

1.6.5 ORDER OF ARRAY STORAGE

Arrays are stored in ascending storage locations, with the value of the first of their subscripts increasing most rapidly and the value of the last increasing least rapidly.

The following list shows the order of a three-dimension array A(3,2,3). The first subscript varies from 1 to 3, the second varies from 1 to 2, the third varies from 1 to 3.



Array allocation is discussed further 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 array.

Given DIMENSION A(L, M, N), the location of A(i, j, k), with respect to the first element of the array, is given by A + (i-1+L*(j-1+M*(k-1)))*E.

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) where A is real, the location of A(2,2,3) with respect to A(1,1,1) is:

LocnA(2,2,3) = LocnA(1.1,1) + (2-1+3*(2-1+3*(3-1)))*1 = LocnA+22

1.6.6 SUBSCRIPTED VARIABLES

A subscripted variable is an alphanumeric identifier that is the name of an array followed by up to three subscript expressions representing a single element within the array. The elements of a subscript expression are separated by commas and the expression is enclosed in parentheses. Subscript expressions may be any legal arithmetic expression. If the number of subscript expressions used in a reference is less than the declared dimensionality, the compiler assumes missing subscripts have a value of one (see examples below). If the subscript list does not appear, all subscript expressions are assumed to be one, and an informative diagnostic is issued.

If the subscript expression is not integer, the value will be truncated to integer.

FORTRAN Extended 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 $A(I, J, K)$

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

$A(I)$ implies $A(I, 1, 1)$

A implies $A(1, 1, 1)$†

Similarly for

$A(D_1, D_2)$

$A(I, J)$ implies $A(I, J)$

$A(I)$ implies $A(I, 1)$

A implies $A(1, 1)$ †

and for $A(D_1)$

$A(I)$ 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 $A(I, J)$. Diagnostics occur if this is attempted.

---

† Except in input/output lists and DATA statements.

EXPRESSIONS 2

An expression is a constant, variable (simple or subscripted), function reference, or any combination of these separated by operators and parentheses. The four kinds of expressions in FORTRAN are: arithmetic and masking (Boolean) expressions which have numerical values, and logical and relational expressions which have truth values. Each kind of expression is associated with a group of operators and operands.

2.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, complex, octal, or Hollerith.

Arithmetic Operators		Arithmetic Operands
+	addition	Constants
-	subtraction	Variables (simple or subscripted)
*	multiplication	Evaluated functions
/	division	
**	exponentiation	

Any unsigned constant, variable, or function reference is an arithmetic expression. If X is an expression, then (X) is an expression. If X and Y are expressions, then the following are expressions:

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

An expression may not contain adjacent operators, such as X +/ Y. Omission of an operator, as for implied multiplication (X) (Y), for instance, is not valid and results in a compiler diagnostic.

The mode of an arithmetic expression is determined by the type specifications of the variables in the expression. The following table indicates how the mode is determined from the possible combinations of variables.

Table 1. Mixed Mode Arithmetic Expressions

+ - * /	Hollerith	Integer	Real	Double Precision	Complex	Octal
Integer	Integer	Integer	Real	Double Precision	Complex	Integer
Real	Real	Real	Real	Double Precision	Complex	Real
Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Precision
Complex	Complex	Complex	Complex	Complex	Complex	Complex
Octal	Integer	Integer	Real	Double Precision	Complex	Integer
Hollerith	Integer	Integer	Real	Double Precision	Complex	Integer

The following examples are valid expressions:

```
A
3.14159
B + 16.427
(XBAR +(B(I,J+I,K) /3))
-(C + DELTA * AERO)
(B - SQRT(B**2*(4*A*C)))/(2.0*A)
GROSS - (TAX*0.04)
TEMP + V(M,MAXF(A,B))*Y**C/ (H-FACT(K+3))
```

The arithmetic operator denoting exponentiation (**) may be used to combine constants, variables, expressions, and subscripted variables. Rules governing the types of variables and constants used in the exponentiation operation are given on the following page:

Base	Exponent	Result
Integer	Integer	Integer
	Real	Real
	Double Precision	Double Precision
	Complex	Complex
Real	Integer	Real
	Real	Real
	Double Precision	Double Precision
	Complex	Complex
Complex	Integer	Complex
Double Precision	Integer	Double Precision
	Real	Double Precision
	Double Precision	Double Precision
	Complex	Complex

The following examples illustrate how constants, variables, and expressions may be combined using the arithmetic operator, **.

Examples:

Expression	Type	Result
CVAB**(I-3)	Real**Integer	Real
D**B	Real**Real	Real
C**I	Complex**Integer	Complex
BASE(M,K)**2.1	Double Precision **Real	Double Precision
K**5	Integer** Integer	Integer
314D-02** 3.14D-02	Double Precision **Double Precision	Double Precision

2.2 RELATIONAL EXPRESSIONS

A relational expression has the value true or false; it contains two arithmetic expressions separated by a relational operator. The types of operands may be combined in the same manner as defined for arithmetic operators. Only the real part of complex elements are compared by relational operators, except for .EQ. and .NE.

Relational operators indicate comparison operations between operands and are enumerated below:

.EQ.	Equal to ($=$)
.NE.	Not equal to ($\neq$)
.GT.	Greater than ($>$)
.GE.	Greater than or equal to ($\geq$)
.LT.	Less than ($<$)
.LE.	Less than or equal to ($\leq$)

A relational expression has the form:

a_1 op a_2

The a_i are arithmetic expressions; op is an operator belonging to the above set.

A relation is true if $\underline{a}_1$ and $\underline{a}_2$ satisfy the relation specified by op; otherwise it is false. A false relational expression is assigned a positive value; a true relational expression is assigned a negative value. Relations are evaluated as illustrated in the relation p.EQ.q, which is equivalent to the question: Does p - q = 0? The difference is computed; and if it is zero, the relation is true; if the difference is not zero, the relation is false. Relational expressions are converted internally to arithmetic expressions according to the rules of mixed-mode arithmetic (Table 1). These expressions are evaluated to produce the truth value of the corresponding relational expressions.

The order of dominance of the operand types within an expression is the order stated for mixed mode arithmetic expressions.

In relational expressions, +0 is considered equal to -0.

a_1 op a_2 op a_3... is not a valid expression. The relations a_1 op a_2, a_1 op (a_2) are equivalent.

Examples:

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

2.3 LOGICAL EXPRESSIONS

A logical expression is formed with logical operators and logical elements and has the value true or false. (The values have the same internal representation as for relational expressions, section 2.2.)

Logical Operators		Alternate Form
.OR.	Logical disjunction	.O.
.AND.	Logical conjunction	.A.
.NOT.	Logical negation	.N.

A logical expression has the general form:

L_1 op L_2 op L_3...

L_i are logical variables, logical constants, logical functions, logical expressions enclosed in parentheses, or relational expressions; and op is the logical operator .AND. indicating conjunction or .OR. indicating disjunction.

The logical operator that indicates negation appears in the form:

.NOT. L_1

Each expression is evaluated by scanning from left to right, with logical operations being performed according to the following hierarchy of precedence.

first	.NOT.
then	.AND.
then	.OR.

A logical variable, logical constant, or a relational expression is, in itself, a logical expression. If L_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 op op L_2 is never legitimate. However, .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....)

L_1 .OR. (.NOT....)

.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:

1. $B - C \leq A \leq B + C$
   is written B - C .LE. A .AND.A.LE.B+C

2. FICA greater than 176.0 and PAYNMB equal to 5889.0
   is written FICA .GT. 176.0 .AND. PAYNMB .EQ.5889.0

3. An expression equivalent to the logical relationship ($P \rightarrow Q$) may be written in two ways:

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

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

2.4 MASKING EXPRESSIONS

Masking expressions consist of masking operators and elements; they resemble logical operations in appearance only.

In a masking expression, 60-bit logical arithmetic is performed bit-by-bit on the operands within the expression. The operands may be any type variables, constants, or expressions, other than logical. No mode conversion is performed during evaluation. If the operand is complex or double precision, operations are performed on the real part, or higher order word. Although the masking operators are identical in appearance to the logical operators, their meanings are different. They are listed according to hierarchy. The following definitions apply:

.NOT.	or	.N.	bit-by-bit logical negation
.AND.	or	.A.	bit-by-bit logical multiplication
.OR.	or	.O.	bit-by-bit logical addition

The operations are described below:

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

If B_i are masking expressions, variables or constants of any type other than 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.

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

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

Masking expressions must not contain logical operands.

Examples:

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

.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

2.5 EVALUATION OF EXPRESSIONS

Evaluation of expressions is generally from left to right with the precedence of the operators and parentheses (the deepest nested parenthetical subexpression is evaluated first) controlling the sequence of operation. The precedence of operators for arithmetic evaluation is shown below:

**	exponentiation	class 1
/	division	class 2
*	multiplication	class 2
+	addition	class 3
-	subtraction	class 3
relationals		class 4
.NOT.		class 5
.AND.		class 6
.OR.		class 7

(Function references may be considered to be class 1.)

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

All function references and exponentiation operations which are not evaluated inline are evaluated prior to other operations.

When writing an integer expression, it is important to remember not only the left-to-right scanning process but also if dividing an integer quantity by an integer quantity yields a remainder the result will be truncated; thus 11/3 = 3.

An array element name (a subscripted variable) used in an expression requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by the evaluation of the actual arguments or subscripts.

The evaluation of an expression with any of the following conditions is undefined:

Negative-value quantity raised to a real, double precision, or complex exponent

Zero-value quantity raised to a zero-value exponent

Infinite or indefinite operand

Element for which a value is not mathematically defined, such as division by zero

If the error traceback option is selected on the FTN card (Appendix C), the first three conditions will produce informative diagnostics.

In the following examples, R indicates an intermediate result in evaluation. A**B/C+D*E*F-G is evaluated:

$A^{**}B \rightarrow R_1$
$R_1/C \rightarrow R_2$
$D*E \rightarrow R_3$
$R_3*F \rightarrow R_4$
$R_2-G \rightarrow R_5$
$R_4+R_5 \rightarrow R_6$ evaluation completed

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

$A^{**}B \rightarrow R_1$
$C+D \rightarrow R_2$
$R_1/R_2 \rightarrow R_3$
$E*F \rightarrow R_4$
$R_4-G \rightarrow R_5$
$R_3*R_5 \rightarrow R_6$ evaluation completed

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

$COS(Z) \rightarrow R_1$
$R_1{}^{**}2 \rightarrow R_2$

(Evaluation of the index function)

$R_2*C(I,J+2) \rightarrow R_3$
$R_3+H(13) \rightarrow R_4$ evaluation completed

The following are examples of expressions with embedded parentheses:

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

$C/D \rightarrow R_1$

$R_1-E \rightarrow R_2$

$B+R_2 \rightarrow R_3$

$A*R_3 \rightarrow R_4$ evaluation completed

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

$SIN(X) \rightarrow R_1$

$R_1+1. \rightarrow R_2$

$A*R_2 \rightarrow R_3$

$R_3-Z \rightarrow R_4$

$E+F \rightarrow R_5$

$D-R_5 \rightarrow R_6$

$C*R_6 \rightarrow R_7$

$R_4/R_7 \rightarrow R_8$ evaluation completed

ASSIGNMENT STATEMENTS 3

Statements are classified as executable or nonexecutable; executable statements specify actions. Assignment statements are executable. They assign values with four types of operations; arithmetic, logical, assign (Chapter 4), and masking.

3.1 ARITHMETIC ASSIGNMENT

The general form of the arithmetic assignment statement is v = e, where v is a variable, simple or subscripted, other than logical; and e is an arithmetic expression. The = indicates that v is assigned the value of the evaluated expression e. Mode conversion occurs if v is of a type different from e.

Examples:

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

Several variables may be assigned the value of the same expression with the following form:

$v_1=v_2=\ldots v_m=e$

The value of expression e is converted to the type of v_m and stored; v_m is then converted to the type of v_{m-1} and stored. The process is repeated until a value is stored in v_1.

Example:

```
      :
RATE=2.0
DATA=6.9
      :
DATA=DATA1=LDATA=DATA2=DATA*RATE
```

The variable, DATA2, equals 13.8 from the expression DATA*RATE. LDATA equals 13 by real-to-integer conversion. DATA1 equals 13.0 by integer-to-real conversion; then DATA equals 13.0 by real-to-real assignment.

MIXED-MODE

The type of an evaluated expression is determined by the type of the dominant operand; however, this does not restrict the types that identifier v may assume. (v may not be logical). A complex expression may replace v, even if v is real. TABLE 2 on page 3-4 shows the v = e relationship for all standard modes. The mode of v determines the mode of the statement.

Examples:

Given:

C_i, A_1	Complex
D_i, A_2	Double
R_i, A_3	Real
I_i, A_4	Integer

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

2. $A_3=C_1$ 4.4=(4.4,2.1)

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

3. $A_3=C_1*(0.,-1.)$ 2.1=(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.

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

5. A_2=D**2*(D_2+(D_3*D_4)) + (D_2*D_1*D_2)

49.68=2.0D0**2*(3.2D0+(4.1D0*1.0D0))+(3.2D0*2.0D0*3.2D0)

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

3.2 LOGICAL ASSIGNMENT

In the general form of the logical assignment statement,

v = e

v is a logical variable or subscripted variable, and e is a logical 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
    5 A = .N.(A.A..N.B).AND.(C.O.D)
      LGA = .NOT.LGB
 2109 LGC = E.OR.LGC.OR.LGB.OR.LGA.OR.(A.AND.B)
```

A multiple replacement statement of the following form is also allowed in logical assignment statements:

$v_1 = v_2 = \ldots v_m = e$

3.3 MASKING ASSIGNMENT

In the masking assignment statement, v = e, e is a masking expression, v is a variable name and may be of any type other than logical. During the assignment, no mode conversion occurs, and the value of the expression is assigned to the first word of v if the type is double precision or complex with the least significant or imaginary part set to zero.

Examples:

```
      INTEGER I, J, K, L, M, N(16)
      REAL B, C, D, E, F(15)

      N(2) = I.AND.J
      B = C.AND.L
   84 F(J) = I.OR..NOT.L.AND.F(J)
      N(1) = I.O.J.O.K.O.L.O.M
      I = .N.I
```

A multiple replacement statement of the following form is also allowed in masking assignment statements:

$$v_1 = v_2 = \ldots v_m = e$$

Table 2 enumerates the assignment of e to v. These rules apply only for arithmetic assignment statements.

Table 2. Rules for Assignment of e to v

v Type	e Type	Assignment
Integer	Integer	Assign
Integer	Real	Fix and Assign
Integer	Double Precision	Fix and Assign
Integer	Complex	† Fix and Assign Real Part
Real	Integer	Float and Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate and Real Assign
Real	Complex	† Assign Real Part
Double Precision	Integer	DP Float and Assign
Double Precision	Real	Real Evaluate, DP Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	† DP Float Real Part and Assign
Complex	Integer	† Float and Assign to Real Part, I
Complex	Real	† Assign Real Part, I
Complex	Double Precision	† DP Evaluate and Real Assign to Real Part, I
Complex	Complex	Assign

†Prohibited combination under USASI FORTRAN (but permitted in FORTRAN Extended).

Assign indicates transmission of resulting value, without change, to entity.

Real Assign indicates transmission to entity, of as much precision as a real value can contain.

DP Evaluate indicates evaluation of the expression according to rules of arithmetic expression evaluation.

Fix indicates truncation of any fractional part of the result and transformation to an integer value.

Float indicates transformation to a real value.

DP Float indicates transformation to a double precision value retaining, in the process, as much precision as a double precision value can contain.

Real Part refers to the real portion of the complex value.

I indicates the imaginary part of the complex value is set to zero.

CONTROL STATEMENTS 4

Control statements alter the sequence of operations or affect the number of iterations of a program section. Control statement labels must be associated with executable statements within the same program unit. Control may not be transferred to a non-executable statement. See appendix F.

4.1 GO TO STATEMENTS

UNCONDITIONAL GO TO

GO TO k

When this statement is executed, control transfers to the statement identified by k.

Example:

```
GO TO 100
GO TO 9
```

4.2 ASSIGNED GO TO

ASSIGN k to i

k is a statement label and i is an integer variable. Execution of this statement and subsequent execution of an assigned GO TO statement using the value i causes the statement k to be executed next. The label must refer to an executable statement in the same program unit containing the ASSIGN statement. k must be the label of an executable statement.

The integer variable i, once used in an ASSIGN statement, may not be referenced in any statement other than an assigned GO TO or an ASSIGN statement until it has been defined in a replacement statement.

Example:

```
        .
        .
        .
    ASSIGN 10 TO KLOK
        .
        .
        .
15  GO TO KLOK,(3, 12, 10, 20)
        .
        .
        .
12  CC = D+E -2*(F/G)
        .
        .
        .
10  D = SQRT(B**C*(1-E))
    ASSIGN 20 TO KLOK
    GO TO 15
        .
        .
        .
20  E = A+1.5
```

ASSIGNED GO TO

GO TO i, $(k_1, k_2, \ldots, k_n)$

i is an integer variable, and k_i are statement labels; i must contain the value assigned by a preceding ASSIGN statement and it must be one of the statement labels in the list. At execution, control transfers to statement identified by k. If the value i is defined by other than an ASSIGN statement, a transfer is made to the absolute memory address represented by the low order 18 bits of i.

Example:

```
        .
        .
        .
    ASSIGN 26 TO INDEX
        .
        .
        .
10  GO TO INDEX, (3,45,26,78,6)
26  BASE (I) = BASE (I+1)*FACT*(CONST**2.0)
    ASSIGN 45 TO INDEX
    GO TO 10
        .
        .
        .
```

COMPUTED GO TO

GO TO ($k_1, k_2, \ldots, k_n$), i

k_i are the statement labels and i is a variable. This statement acts as a many-branch GO TO; i is preset or computed prior to its use in the GO TO statement. Control transfers to k_i, if $1 \leq i \leq n$. If i is less than one or greater than n, a fatal error occurs. The comma separating the statement number list and the index is optional. i must not be specified by an ASSIGN statement.

Example:

```
      .
      .
      .
I=2
N=2
      .
      .
      .
N=N*I
      .
      .
      .
GO TO (100,101,18,102,103)N
```

Control transfers to the statement numbered 102.

Example:

```
ISWICH=1
GO TO (10,20,30) ISWICH(control transfers to 10)
      .
      .
      .
KSWICH=ISWICH+1
GO TO (11,41,31),KSWICH(control transfer is to statement 41)
```

Another form of the statement may be used where i is replaced by e:

GO TO ($k_1, k_2, \ldots, k_n$), e

The value of e is truncated and converted to integer and used in place of i. Control transfers to the statement identified by the label, k_j; where j is the integer value of e at the time of execution. If the value of e is less than one, it is treated as equal to one; if it is greater than n, it is treated as equal to n. The comma before e is optional.

Examples:

1.
```
      .
      .
      .
BRANCH=2.3
INDEX=4
      .
      .
      .
GO TO (23,33,43,53,63),INDEX*BRANCH
```

Control transfers to statement 63 since the integer part of the evaluated expression, INDEX*BRANCH, equals 9 and there is no ninth branch.

2.
```
      .
      .
      .
K=2
X=4.6
      .
      .
      .
GO TO (10,110,11,12,13),X/K
```

Control is transferred to statement 110 since the integer part of the expression X/K equals 2.

4.3 IF STATEMENTS

ARITHMETIC IF THREE-BRANCH

IF (e) k_1, k_2, k_3

e is an arithmetic expression of type integer, real, double precision, or complex, and k_i are statement labels. For complex, only the real part is used in selecting the branch. Execution of the statement results in evaluation of e and transfer of control as follows:

$e < 0$ to statement k_1

$e = 0$ to statement k_2

$e > 0$ to statement k_3

Example:

```
      I = -2
      K = 1
      J = 3
        .
        .
        .
 1    IF (I*K*J)2,3,4    (control transfers to 2)
 2    LDD=LDD+1
      GO TO (40,50,60)LDD
40    IF (X*Y*SIN(X))11,12,13
        .
        .
        .
```

ARITHMETIC IF TWO-BRANCH

A second form of the Arithmetic IF statement; an arithmetic two-branch IF is allowed.

IF (e) k_1, k_2

e may be a masking or arithmetic expression; e is evaluated and control is transferred as follows:

$e \neq 0$	to statement k_1
$e = 0$	to statement k_2

Example:

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

LOGICAL IF

IF (e) s

e is a logical expression and s is any executable statement except a DO statement or another logical IF statement. If the value of e is false, statement s is treated as if it were a CONTINUE statement. If the value of e is true, s is executed.

Example:

```
        :
        :
      B4=DATA(I)
        :
        :
      YMAX=B(ILAST)
      YMIN=B(IFRST)
        :
        :
   16 IF (B4.GE.YMIN.AND.B4.LE.YMAX) GO TO 109
  101 INDEX=INDEX+1
      GO TO 110
  109 KDEX=KDEX+1
        :
        :
```

If B4 is satisfied by the condition, YMIN $\leq$ B4 $\leq$ YMAX, control transfers from statement 16 to 109. If the condition is not satisfied, execution resumes at statement 101.

LOGICAL IF TWO-BRANCH

Another form of the logical IF may be a two-branch statement:

IF (e) k_1, k_2

If the logical statement is true, the statement identified by statement label $\underline{k}_1$ is executed next, if false the statement $\underline{k}_2$ is executed.

4.4 DO STATEMENT

The DO statement makes it possible to repeat a sequence of statements and change the value of an integer control variable during the repetition. A DO statement takes one of the forms:

DO n i=m_1, m_2, m_3 or DO n i=m_1, m_2

The executable statement labeled $\underline{n}$ is the terminal statement of the sequence to be repeated and must physically follow and be in the same program unit as the DO statement.

Example:

```
      :
      :
      DO 100L=300,400
      IF(B(L)) 101,100
101   B(L-100)=B(L)
100   CONTINUE
      :
      :
```

Statement n (100 in the example) may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, DO, two-branch logical IF, or a logical IF followed by any of the preceding statements.

The simple integer variable i is the control variable; m_i are the indexing parameters; m_1 is the initial value of i, m_2 is the terminal value of i, and m_3 is the incrementing parameter. m_i may be either integer constants or simple integer variables. If m_3 is not specified, a value of one is implied. At execution of the DO statement, m_1, m_2 and m_3 must be greater than zero. The range of each DO contains all executable statements between and including the first executable statement after the DO and the terminal statement identified by n.

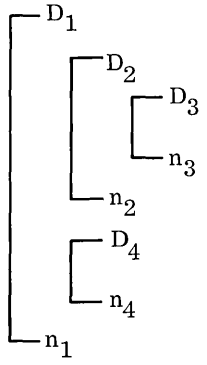
DO NESTS

When a DO loop contains another DO loop, the grouping is called a DO nest. Nesting may be to 50 levels. Either the last statement of a nested DO loop must be the same as the last statement of the outer DO loop or it must occur before it. If $D_1, D_2, \ldots D_m$ represent DO statements where the subscripts indicate that D_1 appears before D_2 appears before D_3 and $n_1, n_2, \ldots, n_m$ represent the corresponding limits of the D_i, then n_m must appear at or before n_{m-1}.

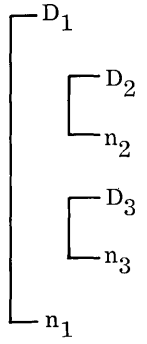
Examples:

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

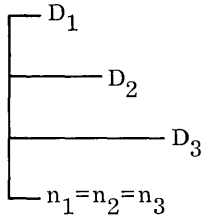
a.



b.



c.



The preceding diagrams would be coded as follows:

a.
```
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
```

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

c.
```
DO 5 I=1,5
DO 5 J=I,10
DO 5 K=J,15
      :
5 A = B*C
```

DO LOOP EXECUTION

The loop defined by a DO statement is executed as follows:

1. The control variable $\underline{i}$ is assigned the value represented by the initial parameter $\underline{m}_1$. The value of $\underline{m}_1$ should be less than or equal to the value of the terminal parameter $\underline{m}_2$; otherwise, the DO loop is executed only once. (The control variables of each nested DO loop must be unique.)

2. The range of the DO is executed.

3. After the DO is executed, the control variable is incremented by the value $\underline{m}_3$ (or by one if $\underline{m}_3$ is not specified).

4. If the value of the control variable $\underline{i}$ after it is incremented by $\underline{m}_3$ is less than or equal to the value of the terminal parameter $\underline{m}_2$, execution of the range of the DO loop is repeated. When the value $\underline{i}$ is greater than the value of $\underline{m}_2$, the DO has been satisfied and the control variable $\underline{i}$, becomes undefined (the value of $\underline{i}$ may be greater, less than or = to $\underline{m}_2$ at the termination of the loop execution, therefore its value cannot be assumed).

5. If the DO is nested, the control variable $\underline{i}$ of the next outer DO is incremented by $\underline{m_3}$ and execution continues repeating steps 4 and 5 until all the DO statements referencing this terminal statement are satisfied. After the last DO is satisfied, execution continues with the first executable statement following its terminal statement.

6. If m_1, m_2, or m_3 are constants which exceed $2^{17}-2$, a diagnostic notes the error and the control variable is used modulo $2^{17}-1$ for iteration of the DO loop.

   Before the DO is satisfied, an exit may be made from its range through an IF or a GO TO statement. In this case, the control variable retains the value last assigned to it before the exit.

   Example:

```
          .
          .
          .
      DO 20 I=1,200
      IF(I-3) 20,10,10
20    CONTINUE
10    I9=I
```

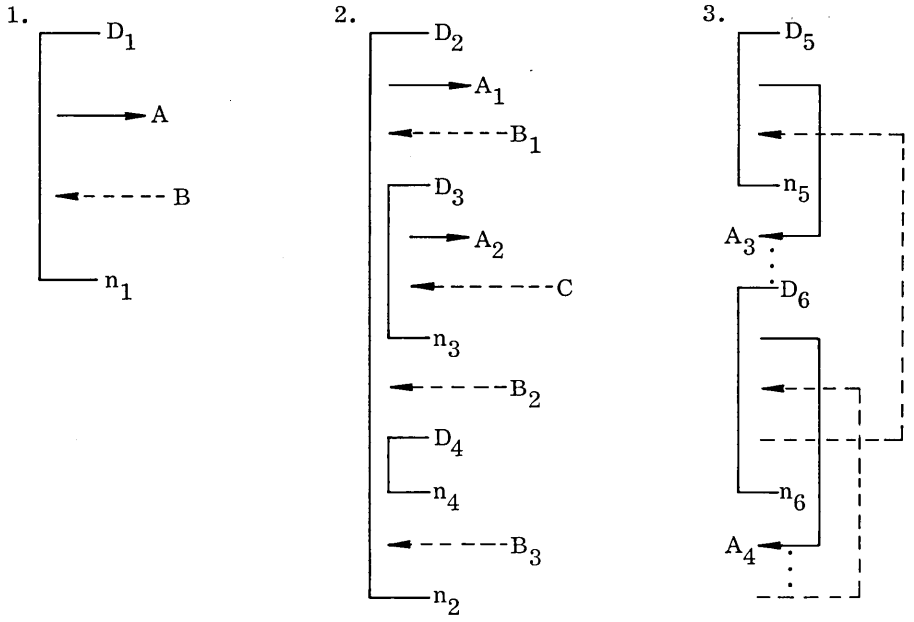
   An exit from the range of the DO is made to statement 10 when the value of the control variable I is equal to 3. The value of the integer variable, I9 is equal to 3, since the last value assigned to I before the exit from the DO range was 3.

A DO has an extended range if both of the following conditions are satisfied:

1. A GO TO or an IF statement within the range of a DO nest transfers control outside the nest

2. A GO TO statement or an IF statement outside the nest causes control to re-enter a DO loop or nested set of DO loops as illustrated below.

Examples:



Example 1 shows an exit at point A. Any re-entry into D_1 may be made as illustrated at point B or at any subsequent point within the indicated loop.

Example 2 shows three nested loops with D_3 and D_4 being parallel. An exit is made at point A_1, re-entry into D_2 may be made at points B_1, B_2, or B_3. However, re-entry cannot be made into D_3 or D_4 because the control variables for those loops have not been defined. If an exit is made from point A_2, re-entry may be made at C, B_1, B_2 or B_3 but not at any other points within the other loops.

The third example illustrates the capability of specifying an extended range DO loop within the extended range of another loop. Loop D_5 has an extended range which is entered at point A_3; the loop D_6, which also has an extended range beginning at point A_4, is contained within the extended range of D_5.

If both conditions are satisfied, the extended range is defined as all statements that may be executed between pairs of control statements, the first of which satisfies condition 1 and the second of which satisfies condition 2. A GO TO or an IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO.

The control variable $\underline{i}$ and the parameters $\underline{m}_1$, $\underline{m}_2$, and $\underline{m}_3$ may not be redefined during execution of the immediate or extended range of that DO. When parameters are redefined during execution, the results are unpredictable. An informative diagnostic is issued for redefinition during an immediate range.

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 IF statement in the nest, except in the range of the innermost DO.

Example:

```
      DO 10 J=1,50
      DO 10 I=1,50
      DO 10 M=1,100
          .
          .
          .
      GO TO 10
          .
          .
          .
10    CONTINUE
```

When the IF statement is used to bypass several inner loops, different terminal statements for each loop are required.

Example:

```
      DO 10 K=1,100
      IF(DATA(K)-10.)20,10,20
20    DO 30 L=1,20
      IF(DATA(L)-FACT*K-10.)40,30,40
40    DO 50 J=1,5
          .
          .
          .
      GO TO (101,102,50),INDEX
101   TEST=TEST+1
      GO TO 104
103   TEST=TEST-1
      DATA(K)=DATA(K)*2.0
          .
          .
          .
50    CONTINUE
30    CONTINUE
10    CONTINUE
          .
          .
          .
      GO TO 104
102   DO 109 M=1,3
          .
          .
          .
109   CONTINUE
      GO TO 103
104   CONTINUE
```

CONTINUE

CONTINUE

This statement is most frequently used as the last statement of a DO loop to provide a loop termination when a GO TO or IF would normally be the last statement of the loop. If CONTINUE is used elsewhere in the source program it acts as a do-nothing instruction and control passes to the next sequential program statement.

Example:

```
      .
      .
      .
    DO 10K = 1,200
    DATA(K)=DATB(K+1)
10  CONTINUE
```

4.5 CALL

The CALL statement, which transfers control to a subroutine subprogram, may take one of the following forms:

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

CALL s

CALL s $(a_1, a_2, \ldots, a_n)$, RETURNS $(b_1, b_2, \ldots, b_m)$

CALL s, RETURNS $(b_1, b_2, \ldots, b_n)$

s is the name of a subroutine and a_i are actual arguments which correspond to dummy arguments specified in the subroutine subprogram. b_i parameters indicate labels of statements in the current calling program or subprogram. The total number of parameters, $a_i + b_i$, should not exceed 63.

The arguments (a_i) appearing in the statement may be constants, variables, array element names, array names, the name of an external procedure, etc. (see p. 9-4). These arguments must correspond in number, order and type with those specified in the SUBROUTINE statement (see chapter 9 for an explanation of this statement).

The parameters b_i must be specified with the RETURNS if alternate exits are taken from the subroutine. If alternate exits are not taken, this specification may be omitted, and control returns to the statement immediately following the CALL. These parameters must also correspond to similar parameters specified in the subroutine.

The return of control from the designated subroutine completes the execution of the CALL statement.

Example:

```
      PROGRAM MAIN (INPUT, OUTPUT)
          :
          :
   10 CALL XCOMP(A, B, C), RETURNS(101, 102, 103, 104)
          :
          :
  101 CONTINUE
          :
          :
      GO TO 10
  102 CONTINUE
          :
          :
      GO TO 10
  103 CONTINUE
          :
          :
      GO TO 10
  104 CONTINUE
      END

      SUBROUTINE XCOMP (B1, B2, G), RETURNS (A1, A2, A3, A4)
      IF(B1*B2-4.159) 10, 20, 30
   10 CONTINUE
          :
          :
      RETURN A1
   20 CONTINUE
          :
          :
      RETURN A2
   30 CONTINUE
          :
          :
      IF (B1) 40, 50
   40 RETURN A3
   50 RETURN A4
      END
```

RETURN

RETURN or RETURN a

a is a formal parameter (as indicated in the RETURNS list).

Example:

```
      SUBROUTINE XYZ, (P, T, U), RETURNS(A, B, C)
      IF (P*T*U) 1, 2, 3
1     CONTINUE
          .
          .
          .
      RETURN A
2     CONTINUE
          .
          .
          .
      RETURN B
3     RETURN C
      END
```

The statement, RETURN a, can appear only in a subroutine subprogram. Execution of this statement returns control to the statement number corresponding to a in the RETURN list.

A RETURN statement marks the logical end of a procedure (subroutine or function) subprogram and may appear only in a procedure subprogram. In a subroutine subprogram, a RETURN statement returns control to the next executable statement following the CALL statement of the current calling program. In function subprograms, a RETURN statement returns control to the statement containing the function reference.

4.6 PROGRAM CONTROL

STOP

STOP n or STOP

n is a string of 1-5 octal digits.

When a STOP statement is encountered, n is displayed in the dayfile, the executable program terminates and control returns to the monitor. If n is omitted, blanks are implied.

PAUSE

PAUSE n or PAUSE

n is a string of 1-5 octal digits.

When a PAUSE statement is encountered, the executable program halts and PAUSE n appears as a dayfile message on the display console. The operator can continue or terminate the program with an entry from the console. The program continues with the next statement. If n is omitted, blanks are implied.

END

END

This must be the final statement and marks the physical end of the program or subprogram. It is executable in the sense that it effects termination of a main program or acts as a RETURN in a SUBROUTINE or FUNCTION, but it may not be labeled.

INPUT/OUTPUT STATEMENTS 5

The READ and WRITE input/output statements cause information to be transferred between internal storage and external devices.

5.1 MODES OF INPUT/OUTPUT

Input and output can be formatted or unformatted. Formatted information consists of strings of characters acceptable to the FORTRAN processor. Unformatted information consists of strings of binary word values in the form in which they normally appear in storage. The transmission of formatted information is always associated with a FORMAT statement, as described in chapter 6. Additionally, NAMELIST may be used for input/output as discussed in section 5.6.

5.2 I/O LISTS

The input list specifies the names of variables and array elements to which information is transmitted from the external device. The output list specifies the variables and array elements whose values are transmitted to the external device. Both lists may take any of the following forms.

If no list appears on input, a record is skipped. Only Hollerith information from the FORMAT statement can be output with a null (empty) output list.

A simple list consists of a variable name, an array name, an array element name, or a DO-implied list.

If an array name without any subscripts appears in a list, the entire array (not just the first word of the array) is read or written.

Multiple simple lists may appear, separated by commas, each of which may be enclosed in parentheses, such as: (...),(...).

A DO-implied list is a simple list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification takes one of the following forms:

$i = m_1, m_2, m_3$ or $i = m_1, m_2$

The elements i, m_1, m_2 and m_3 have the same meaning for the DO statement. The range of DO-implied specification is that of the DO-implied list. For the input lists, i, m_1, m_2 and m_3 may appear within that range only as subscripts.

Elements of a list are specified in order of occurrence from left to right. The elements of a DO-implied list must be specified for the initial cycle of the implied DO.

5.3 READ/WRITE STATEMENTS

The parameters used with the READ/WRITE statements are defined as follows:

- u Identifies the input/output unit; an integer constant or a simple integer variable.
- f Identifies the format specification; a FORMAT statement label or an array name. If f is a statement label, the statement must appear in the same program unit as the input or output statement.
- k Input/output list indicating the data to be transferred.

5.4 FORMATTED INPUT/OUTPUT

The statements discussed in this section pertain to the transmission of data according to a FORMAT specification.

Information processed by the READ and WRITE statements is divided into records. Each time a READ or WRITE is executed at least one record is processed. It is not possible to read several parts of a single record with more than one READ statement.

READ

READ (u,f)k READ (u,f) READ f,k

This statement transmits data from the external device for which the logical unit number is the integer value of u. Information contained on u is scanned and converted in accordance with the format specification identified by f. The values, as a result, are assigned to the element specified by the list, k. However, if the list is omitted, this statement means the next logical record is bypassed (except for the case described on page 6-15 of reading Hollerith characters into an existing H field within a FORMAT statement).

Example:

```
      READ (2,10) (IDAT(I), I = 1,10), C
10    FORMAT (2X, 10(I5,2X),F3.2)
      DO 30 K=1,10
      READ (2,20) (B2(K,J),J=1,5)
20    FORMAT (5(F10.2,1X))
30    CONTINUE
```

INPUT FILE

READ f,k or READ f

This statement results in the input of records from the SCOPE INPUT file. The theory of operation is the same as that described for the formatted READ statement.

Example:

```
      READ 31,NAME,GREEN,HORNET
31    FORMAT (A10,F10.3,E20.2)
```

WRITE

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

The above statements write formatted records on the logical unit specified by u. The parameters have the same meaning as described for the corresponding READ statement. The contents of the resulting records consist of the values of the list items in the order in which they appear in the list. The values represented by the list variables are converted according to the format specification, then transferred to the indicated output unit.

Example:

```
      WRITE (6,10) L1,B1,L2,B2
10    FORMAT (2X,I5,1X,F5.2,I5,F9.3)
      DO 20 J = 1,10
      DO 20 K = 1,10
20    WRITE (4,26) DATA1(J,K), DATA1 (J,K)
26    FORMAT (2X,15H THE VALUES ARE, 2F6.2)
```

If the list k in a formatted WRITE statement is omitted, the contents of the created record are dependent upon the corresponding FORMAT statement.

Example:

```
      WRITE (4,27)
27    FORMAT (32H THIS COLUMN REPRESENTS X VALUES)
```

When the list k is specified for formatted input or output, the corresponding FORMAT declaration must contain at least one conversion specification other than Hollerith.

PRINT/PUNCH

PRINT f,k or PRINT f

The information specified by k is transferred as line printer images to the SCOPE OUTPUT file, 136 characters or less per line in accordance with FORMAT declaration f.

Example:

```
      PRINT 20, DNAME
20    FORMAT (X,A10)
```

When the list designation is omitted, the statement has the form illustrated in the following example:

```
      PRINT 20
20    FORMAT (31H THIS IS THE END OF THIS REPORT)
```

The first character of formatted records is not printed, but is used by the line printer to determine vertical spacing of records on a page. Appendix I, carriage control characters, lists the control options.

PUNCH f, k or PUNCH f

The information specified by k is transferred to the SCOPE PUNCH file as Hollerith images, 80 characters or less per card in accordance with FORMAT declaration f. If the card image is longer than 80 characters, a second card is punched with the excess characters. Omission of k is interpreted the same as for the PRINT statement.

Example:

```
      PUNCH 30, JOHN
30    FORMAT (X,I7)
```

5.5 UNFORMATTED INPUT/OUTPUT

The statements discussed herein transmit data without a FORMAT designation.

READ

READ (u)k or READ (u)

This form of the READ statement is classified as unformatted because of the omission of the f parameter in the statement form. Execution of the statement results in the sequential assignment of values, as they are read, to the variables appearing in the list k. The sequence of values required by the list may not exceed the length of the unformatted record. However, if the list is omitted, this statement serves merely to designate the bypassing of the next logical record; no information is transmitted from the source device.

Examples:

```
READ (30)
READ (31) DATA1, DATA2,IDATA
READ (32) (SUM(K), K=1,100)
READ (33) I,J,K,L,M,N
```

WRITE

WRITE (u) k or WRITE (u)

This form of the WRITE statement creates the next record on the unit identified by u. The contents of the record are the sequence of values specified by the list k.

Examples:

```
WRITE (30) (DATA(I), I=1,100)
WRITE (31) I,J,K,R
WRITE (32) PAY, COST, BAL
```

If the list is omitted from the statement, a null record is written on the output device. A null record is a record which consists of no data but contains all the other properties of a legitimate record.

Example:

```
WRITE (14)
```

5.6 NAMELIST STATEMENT

The NAMELIST statement permits the input and output of character strings consisting of names and values without a format specification.

NAMELIST $/y_1/a_1/y_2/a_2/.../y_n/a_n$

Each y is a NAMELIST group 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 $b_1,b_2,...,b_n$; 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.

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 group 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 in a READ, WRITE, PRINT, or PUNCH statement. The form of the input/output statements used with NAMELIST is as follows:

```
READ (u,x)
READ x
WRITE (u,x)
PRINT x
PUNCH x
```

u is an integer variable or integer constant denoting a logical unit, and x is a NAMELIST group name.

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 the BCD (coded) input/outputs 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:

$v = c$

$a = d_1, \ldots, d_j$

$a(n) = d_1, \ldots, d_m$

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:

```
$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,12.0E1,13,4*14,
Z=(1.,2.),K=16,M=17$
```

produces the following values in memory:

```
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. These names need not be in order in which they appear in the defining NAMELIST statement.

OUTPUT DATA

When a NAMELIST group name is referenced in a WRITE (u,x), PRINT x, or PUNCH x statement, the entire list associated with that name is output as BCD information. Output consists of at least three records. The first record is a $ in column 2 followed by the group identifier x; the last record is a $ in column 2 followed by the letters END. Between these two records are as many records as necessary to output the current values of all variables in the list associated with x.

Each variable or array is output as a separate record, with no data appearing in column 1 of any record. Simple variables are output as v = c. Elements of dimensioned variables are output in the order in which they are stored internally. Logical constants appear as T and F. The data fields are made large enough to include all significant digits.

The records output by a WRITE (u,x) statement may be read by a READ (u,x) statement. The maximum length of a record written by a WRITE (u,x) statement is 130 characters. If unit u is the standard punch unit and a record to be output contains more than 80 characters, a second card is used for the record.

5.7 REWIND

REWIND u

This statement positions unit u at its initial point. If the statement is not applicable to the unit specified or u is at the initial point, the statement has no effect.

Example:

```
REWIND 31
REWIND L
```

5.8 BACKSPACE

BACKSPACE u

Execution of this statement positions unit u so that what had been the preceding user logical record becomes the next record. If the statement is not applicable to the unit specified or unit u is at the initial point, the statement has no effect.

Example:

```
BACKSPACE 40
BACKSPACE K
```

5.9 ENDFILE

ENDFILE u

When this statement is executed, an end-of-file record is written on unit u. The end-of-file record indicates a demarcation of a file.

Example:

```
ENDFILE 31
ENDFILE M
```

5.10 ECS I/O

The following statements result in data transmission between ECS (Extended Core Storage) and central memory.

CALL READEC (a,b,n)

CALL WRITEC (a,b,n)

- a Simple or subscripted variable located in central memory.
- b Simple or subscripted variable located in ECS common block.
- n Integer constant or integer expression.

When either statement is executed, n consecutive words of data are transmitted between central memory and ECS beginning at location a in central memory and b in Extended Core Storage.

5.11 MASS STORAGE I/O

Four object time subroutines control record transmission between central memory and a mass storage device. The references to these routines take the following forms:

CALL OPENMS (u,ix,ℓ,p)

CALL READMS (u,fwa,n,i)

CALL WRITMS (u,fwa,n,i)

CALL STINDX (u,ix,ℓ)

- u Logical unit number.
- ix First word address of the index (in central memory).
- ℓ Length of the index; $\ell \geq 2$ (number of index entries)+1 for a name index; $\ell \geq$ number of index entries+1 for a number index.

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

fwa Central memory address of the first word of the record.

n Number of central memory words to be transferred.

i Record number or the address of a cell containing the record name (left justified display code with binary zero fill, 1 to 7 characters) or number.

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. OPENMS must be called before READMS, WRITMS, and STINDX.

The routines READMS and WRITMS perform the actual transfer of data to or from central memory.

STINDX is called to change the file index to the base specified in the CALL (See Appendix I for further information and examples concerning the use of these routines.)

The random access name must be left justified display code, from 1-7 characters long, with binary zero fill.

The FORMAT statement is used in conjunction with the input/output of formatted records to indicate the manner of converting and editing information between the internal representation and the external character strings.

6.1 FORMAT DECLARATION

FORMAT $(q_1 t_1 z_1 t_2 z_2 \ldots z_{n-1} t_n q_2)$

q series of slashes (optional)

t field descriptor or groups of field descriptors

z field separator

n may be zero

The FORMAT declaration is non-executable and may appear anywhere in the program. It must have a statement label in columns 1-5. FORMAT statements are analyzed for validity by the compiler. Diagnostics are provided.

FIELD DESCRIPTORS

The format field descriptors are:

srEw.d	Single precision floating point with exponent
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
rLw	Logical conversion
rAw	Alphanumeric conversion
rRw	Alphanumeric conversion
rOw	Octal integer conversion
$nHh_1 h_2 \ldots h_n$	Hollerith character control
nX	Intraline spacing
... or ≠...≠	Hollerith string delimiters
Tn	Column tabulation

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

w and n are non-zero integer constants which represent the field width in the external character string. n used with T indicates the beginning column position for subsequent information.

d is an integer constant which represents the number of digits in the fractional part of the external character strings (except for G conversion).

r is the repeat count. It is represented by an optional non-zero integer constant and indicates the repetition factor of the succeeding basic field descriptor.

s is optional and represents a scale factor.

h is one of the characters in the machine character set.

* or ≠ is used to delimit Hollerith strings. (≠ prints as ' on many printers.)

For all descriptors, the field width w or n must be specified. If d is not specified for w.d, it is assumed to be zero.

FIELD SEPARATORS

The two format field separators are the slash (/) and the comma (,). Series of slashes are another form of field separator. Field separators are used to separate field descriptors and groups of field descriptors. The slash is also used to specify demarcation of formatted records.

6.2 CONVERSION SPECIFICATION

Leading blanks are not significant in numeric input conversions; other blanks are treated as zeros. Plus signs may be omitted. An all blank field is considered to be minus zero, except for logical input, where an all blank field is considered to be FALSE. When an all blank field is read with a Hollerith input specification (R or A), each blank character will be translated into a display code 55 octal.

For the E, F, G, and D input conversions, a decimal point in the input field overrides the decimal point specification of the field descriptor.

The output field is right justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field. The number of characters produced by an output conversion must not be greater than the field width. If the field width is exceeded, an asterisk is inserted in the leading position of the field .

Any output which is sent to the line printer uses the first character on the left for carriage control. Thus, the first character is lost and printing begins in the first print position using the second character. This applies only to line printers, not to other output devices.

Iw INPUT

This specification, in conjunction with an input statement, designates a decimal integer constant; field length of w characters. The input field is an optionally signed integer or blank. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. The value is stored right-justified in the specified variable.

Example:

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

Stored Variable:	I	J	K	L	M	N
Input Card:	139bb-15bb18bb7b3b1b4					
Field Width:	3	7	2	3	2	4

Iw OUTPUT

I specification may also be used to indicate the output of decimal integer values. The output quantity occupies w output character positions, right justified:

ba...a

where b is a blank or minus sign if the integer is negative, a's are the digits (maximum 15) of the integer. If the integer is positive, the + sign is suppressed. If the field width w is larger than required, the output quantity is right justified with blank fill to the left. If the field is too short, characters are stored from the right; an asterisk occupies the leftmost position, with excess characters being discarded from the left. If the integer is greater than $2^{48}-1$, an X is printed in the field.

Example:

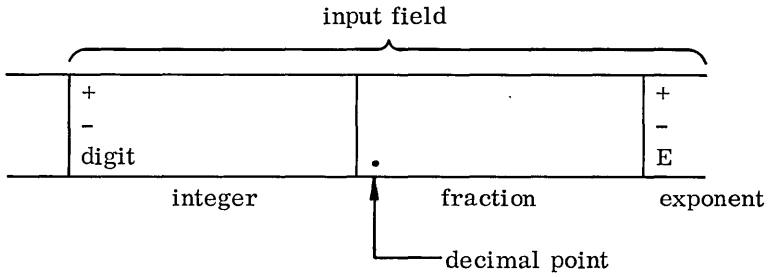
```
      PRINT 10,I,J,K          I contains -3762
10    FORMAT (I8,I10,I5)      J contains +4762937
                              K contains +13

      Result:  bbb-3762bbb4762937bbb13
                  8       10      5
```

Ew.d INPUT

The E specification designates the conversion and storing of a number in the input field as a real number. 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.

Subfield structure of the input field:



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 - followed by an integer constant right adjusted in the field. When it begins with D, or E, a sign 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.

Permissible subfield combinations:

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

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:

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

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

A decimal point in the input field overrides d. The input quantity 3.672+5 read by an E9.d specification is always stored as 3.672×10^{5}. When d does not appear, it is assumed to be zero.

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. The field w includes blanks, significant digits, signs, decimal point, E or D and the 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:

```
      9      5       10
+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, +6.47-01 is read, converted, and placed in location A. Next, -2.36+5 is read, converted, and placed in location B. The number actually desired was -2.36, but the specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input representation under the definitions and restrictions.

Finally, .321E+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 above example illustrates a situation where numbers are incorrectly read, converted, and stored, and yet there is no immediate indication that an error has occurred.

Examples:

Input Field	Ew.d Input 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 $8936. \times 10^{-10+4}$
327.625	E7.3	327.625	No exponent subfield
4.376	E5	4.376	No d in specification
-.0003627+5	E11.7	-36.27	Integer subfield contains - only
-.0003627E5	E11.7	-36.27	Integer subfield contains - only
blanks	Ew.d	-0.	All subfields empty
1E1	E3.0	10.	No fraction subfield; input number converted as $1.x10^{1}$
E+06	E10.6	0.	No integer or fraction subfield; zero stored regardless of exponent field contents
1.bEb1	E6.3	10.	Blanks are interpreted as zeros

Ew.d OUTPUT

Real numbers in storage are converted to the BCD 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:

b.a...a±eee $100 \leq eee \leq 308$

or

b.a...aE±ee $0 \leq ee \leq 99$

b indicates no character position or minus sign; a's are the most significant digits of the value, and eee are the digits in the exponent. If d is zero or no character, the digits to the right of the decimal do not appear as shown above. Field w must be wide enough to contain the significant digits, sign (if negative), decimal point, E, and the exponent. Generally, $w \geq d+6$. Since positive numbers do not require a sign, space need not be reserved for one.

If the field is not wide enough to contain the output value, an asterisk is inserted in the high order position of the field. If the field is longer than the output value, the quantity is right justified with blank fill to the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

Examples:†

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

          Result: b-.673E+02 or bb.673E+02

      PRINT 10,A
10    FORMAT (E13.3)

          Result: bbbb-.673E+02 or bbbbb.673E+02

      PRINT 10,A          A contains -67.32
10    FORMAT (E8.3)       no provision for - sign

          Result: *.67E+02

      PRINT 10,A
10    FORMAT (E10.6)

          Result: *.6732E+02
```

Fw.d INPUT

This specification is the same as Ew.d input specification. It may be used for the transfer of real data that does not contain a decimal exponent.

†In the examples, the use of column 1 for carriage control has been ignored. The results demonstrate the way in which data is converted, not the way the line will appear when printed.

Fw.d OUTPUT

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

ba...a.a...a

b identifies a minus sign or no character position and a's represent the most significant digits of the number.

The number of decimal places to the right of the decimal is specified by d. If d is zero or omitted, digits to the right of the decimal point do not appear. If the number is positive, the + sign is suppressed. If the field is too short to accommodate the number, one asterisk appears in the high-order position of the output field. Field w must be wide enough to contain significant digits, sign (if negative), and a decimal point. If the field is longer than required to accommodate the number, the number is right justified with blank fill to the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

Examples:†

A contains +32.694

```
    PRINT 10,A
10  FORMAT(F7.3)
```

Result: b32.694

```
    PRINT 11,A
11  FORMAT(F10.3)
```

Result: bbbb32.694

A contains -32.694

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

Result: *2.694

A contains .32694

```
    PRINT 13,A,A
13  FORMAT(F4.3,F6.3)
```

Result: .327bb.327

†In the examples, the use of column 1 for carriage control has been ignored. The results demonstrate the way in which data is converted, not the way the line will appear when printed.

Gw.d INPUT

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

Gw.d OUTPUT

The G conversion specifies the transfer of real data where w designates the field length and d denotes the number of significant digits of the value to be represented.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \leq N < 1$	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	sEw.d

The effect of the scale factor is suspended unless the magnitude of the datum to be converted exceeds the range that permits effective use of the F conversion. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

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, w must be ≥ d+6.

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: bb.121E+07

Dw.d OUTPUT

D conversion corresponds to Ew.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. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

b.a···a±eee	$100 \leq eee \leq 308$
or	
b.a···aD±ee	$0 \leq ee \leq 99$

Dw.d INPUT

D conversion corresponds to E conversion except that the list variables must be double precison 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:

```
-6.31675298443D-03+2.718926453147629347752886 9D-09
        18              15              17
```

Ow OUTPUT

O specification is used to output octal integer values. The output quantity occupies w output character 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.

Ow INPUT

Octal integer values are converted under O specification. The field is w characters in length.

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. Embedded blanks are interpreted as zeros.

Example:

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

Input Card: 3737373737666b6644b444-0

10 12 2

In storage:

```
P   0000000003737373737
Q   0000000666066440444
R   7777777777777777777
```

A negative octal number is represented internally in one'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 7777777777777777074.

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

Aw OUTPUT

A 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 represented by leftmost w characters.

Aw INPUT

This specification accepts FORTRAN characters including blanks. The internal representation is 6000 Series 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 BCD word; the remaining spaces are blank filled.

Example:

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

Input card: LUX MENTIS LUX ORBIS

8 8 4

In storage: Q LUXbMENTbb
P ISbLUXbObb
O RBISbbbbbb

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.

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.

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

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,K,L are negative (TRUE) and
      PRINT 5,I,J,K,L        J is positive (FALSE)
    5 FORMAT (4L3)
```

Result: bbTbbFbbTbbT

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. If the first non-blank character is neither T nor F, the field is considered false.

COMPLEX CONVERSIONS

The specification by which a complex variable is read or written requires the designation of two real field descriptors: the first designates the real part, the second the imaginary part. The field descriptors that may be used are: E (Ew.d), F(Fw.d), or G(Gw.d).

Example:

```
INTEGER A
COMPLEX CC                          where  A = 3762
PRINT 20,A,B,CC,D                          B = 833.275
FORMAT (I5,F8.3,E10.4,E9.2,G11.5)  CC = 36.292, -46.73
                                           D = .62534
```

Results: b3762 b833.275 b.3629E+02b-.47E+02 b.62534bbbb

nP SCALE FACTOR

A scale factor that may be used with F, E, G, and D conversions is of the form:

```
nP
nPFw.d
nPEw.d
nPGw.d
nPDw.d
```

n, the scale factor, is a positive (unsigned) or negative integer constant.

A scale factor of zero is established when the format control is initiated; it holds for all F, E, G, and D field descriptors until another scale factor is encountered.

The scale factor n affects conversions as follows:

For F, E, G, and D input conversions (provided no exponent exists) in the external field) and F output conversions: External number = Internal number x 10^n

For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

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.

For G output, the effect of the scale factor is suspended unless the magnitude of the data to be converted exceeds the range that permits effective use of F conversion. If the effective use of the E conversion is required, the scale factor has the same effect as with E output.

Examples:

Using an internal number of 3.1415926538, some output representations with the use of a scale factor are:

Specification	Output Representation
E20.2	.31E+01
1PE20.2	3.14E+00
4PE20.2	3141.59E-03
7PE20.2	3141592.65E-06
-1PE20.2	.03E+02
5PF20.2	314159.27
-2PF20.4	.0314

6.3 EDITING SPECIFICATIONS

nX

This specification permits spacing of input/output quantities; it permits blanks to be inserted in an output record or n characters to be skipped in an input record. The designation of 0X is ignored and bX is interpreted as 1X. In the specification list, a comma following X is optional.

Examples:

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

Result: b7bbbbbbb13.60bbbbbbbb.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

nH

This specification provides for the input or output of 6-bit characters, including blanks, in the form of comments, titles, and headings. An unsigned integer n specifies the number of characters, maximum of 136 to the right of H that are transmitted to the output record; H denotes a Hollerith field; the comma following an 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
      FORMAT (6HbLMAX=,F5.2)        comma is optional
```

produces output record:

bLMAX = b1.50

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:

bTHISbISbAbVARIABLEbHEADING

27 columns

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:

bTHISbISbAbVARIABLEbHEADING

NEW RECORD

The slash (/) indicates the end of the last record anywhere in the specification list. Consecutive slashes may appear and need not be separated from the other list elements by commas. During output, the slash is used to produce blank records. During input, it is used to bypass records. k(/) is equivalent to $/_1/_2,\ldots,/_k$.

Examples:

```
1.      PRINT 10
   10   FORMAT (6X,7HHEADING///3X,5HINPUT,2X,6HOUTPUT)
```

Printout:

```
      bbbbbbHEADING                line 1

                          (blank)  line 2

                          (blank)  line 3

      bbbINPUTbbOUTPUT             line 4
```

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

```
2.      PRINT 10,A,B,C,D
   10   FORMAT (2E10.3/2F7.3)
```

```
In storage:  A   -11.6
             B   .325
             C   46.327
             D   -14.261
```

Printout:

```
      b-.116E+02bb.325E+00
      b46.327-14.261
```

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

Printout:

```
      b-.116E+02bb.325E+00                line 1

                                (blank)   line 2

      b46.327-14.261                      line 3
```

4.
```
      DIMENSION X(3)
      PRINT 15,(X(I),I=1,3)
15    FORMAT (8HbRESULTS2(/)(3F8.2))
```

Resultant lines:

```
bRESULTS                              line 1

                              (blank) line 2

3.62     -4.03     -9.78              line 3
```

The same results may also be obtained by using the statement, PRINT 15,X

... ≠...≠

Hollerith string delimiters are *...* and ≠...≠. All characters (including) blanks) enclosed by a pair of delimiters are read or written. Each character may appear in a field delimited by the other. In an nH delimited specification, the * or ≠ (' for some printers) will be reproduced.

Example:

```
      PRINT 10
10    FORMAT (20X*THISbISbTHEbENDbOFbTHISbRUN*,T52*...HONEST*)
```

Result: (beginning in print position 20)

```
1234567890123456789 ...1234567890123456789012345678901234567890
                    THIS IS THE END OF THIS RUN    ...HONEST
```

Tn

This specification is used as a tabular column selection control. When Tn is used, the format pointer is skipped to column n and the next format specification is processed. n may be any unsigned integer, maximum of 136. If n = zero, column 1 is assumed. (If output is to a line printer, printing is left-shifted one character due to carriage control requirements.)

Using card input, if n > 80 the column pointer is moved to column n but a succeeding specification would read only blanks.

Examples:

1)
```
      PRINT 60
60    FORMAT (T80,*COMMENTS*, T60,*HEADING4*, T40
             *HEADING3*,T20,*HEADING2*,T2,*HEADING1*)
```

Produces the following output: print positions are indicated by the upper line of numbers 1-80.

```
1              19             39             59             79
HEADING1       HEADING2       HEADING3       HEADING4       COMMENTS
```

```
2)      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.

```
3)      READ (20,20)
    20  FORMAT (T10,*COLUMN1*)
```

The first nine characters of the input record are skipped and the next seven are read from input file 20; these seven characters replace COLUMN1, the data in storage.

6.4 REPEATED FORMAT SPECIFICATIONS

FORMAT specifications may be repeated by using an unsigned integer constant repetition factor k as follows: k(spec). For example, to print the array Y:

```
        PRINT 10, (Y(I),I=1,9)
    10  FORMAT (3(3F8.3))
```

is equivalent to:

```
        PRINT 10, (Y(I),I=1,9)
    10  FORMAT (9F8.3)
```

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))
```

If no group repetition factor is specified, a basic group (repetition factor of one) is assumed. If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and 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 specification.

Further groupings may be formed by enclosing field descriptors, field separators, or basic groups within parentheses, and a group repetition factor may be specified for these groupings. The parentheses enclosing the format specification are not considered as group delimiting parentheses.

FORMAT statement specifications may be nested to a depth of two. For instance:

```
10    FORMAT(1H0,3E10.3/(I2,2(F12.4,F10.3))/D28.17)
```

6.5 VARIABLE FORMAT

FORMAT specifications may be indicated at the time of program execution. The specification, including left and right parentheses but not the statement label or the word FORMAT, must be Hollerith data stored in an array. The name of the array containing the specifications may be used in place of the FORMAT statement labels in the associated input/output operation. The array name specifies the location of the first word of the FORMAT information and may appear with or without a subscript.

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 such as:

```
      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)
```

which is the same as:

```
      PRINT 1,(A(I),I=1,2),K,B,C,(LSN(J),J=1,3)
1     FORMAT (2F6.3,I7,2E12.2,3I1)
```

Output statement:

```
PRINT LAIS2,LA,(A(M),M=3,4),A(6),(TEMP(I),I=2,3)
```

which is the same as:

```
      PRINT2,LA,(A(M),M=3,4),A(6),(TEMP(I),I=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)
```

which is the same as:

```
      WRITE(10,10)NUM,VALUE(6)
10    FORMAT(I3,13HMEANbVALUEbIS,F6.3)
```

7.1 BUFFER STATEMENTS

Some of the characteristics of buffered input/output are given below:

1. The mode of transmission (BCD or binary) is tacitly implied by the form of the input/output control statements. In a buffer control statement, parity must be specified by a parity indicator.

2. The input/output 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. Use of an input/output control statement does not return control to the program until completion of the operation. A buffer control statement initiates data transmission, then returns control to the program, permitting the program to perform other tasks while data transmission is in progress. Before buffered data may be used, status of the buffer operation should be checked through use of the UNIT function (see Appendix I). Failure to perform a status check renders the result of the last buffer operation unpredictable.

In the following discussion, the definitions of the indicated parameters are as follows:

- u Logical unit number; an integer constant or variable which may range in magnitude from 1 to 99.
- p Recording mode; an integer constant or variable which may assume the value of zero, designating even parity (coded mode) or 1 indicating odd parity (binary mode).
- A First word address of the block of data to be transmitted.
- B Last word address of the block of data to be transmitted. This address must be greater than or equal to A.

A unit referenced in a BUFFER statement may not be referenced in other statements except REWIND, BACKSPACE and ENDFILE.

BUFFER IN

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

This statement transfers information from unit u in mode p to storage location A through B. Only one logical record is read for each BUFFER IN statement.

BUFFER OUT

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

This statement initiates output of data contained in locations A through B onto unit u. One logical record is written for each BUFFER OUT statement.

A more detailed discussion of these statements is given in Appendix I.

7.2 ENCODE/DECODE STATEMENTS

The ENCODE/DECODE statements are comparable to the BCD 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:

- n Unsigned integer constant or a simple integer variable (not subscripted) specifying the number of characters in the record. n may be an arbitrary number of BCD characters.
- f Statement number or variable identifier representing the FORMAT statement.
- A Identifier of a variable or an array which supplies the starting location of the BCD record.
- k Input/output list.

The first record begins with the leftmost character position specified by A and continues until n BCD characters have been transferred (10 BCD characters per computer word).

Each succeeding record begins with a new computer word, the integral number of computer words allocated for each record is $\frac{n+9}{10}$.

Further information on these statements is given in Appendix I.

ENCODE

ENCODE (n,f,A)k

The list of variables, k, is transmitted according to the FORMAT f and stored, n BCD characters per record, starting at location A. If n is not a multiple of 10, the remainder of the word is blank filled. If the I/O list k and the specification list f translate more than n characters per record, an execution diagnostic occurs.

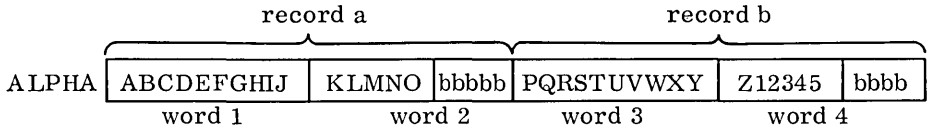
Examples:

```
A(1) = 10HABCDEFGHIJ
A(2) = 10HKLMNO
B(1) = 10HPQRSTUVWXY
B(2) = 10HZ12345
```

1. n = multiple of 10

```
      ENCODE (20,1,ALPHA)A,B
    1 FORMAT (A10,A5/A10,A6)
```

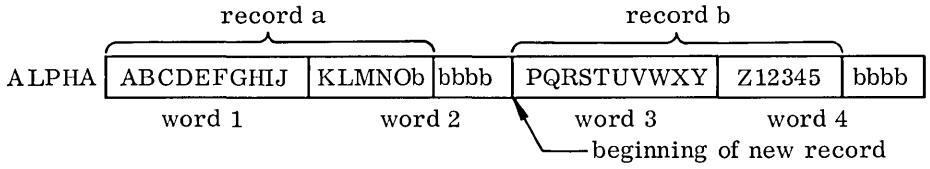
Result:



2. n ≠ multiple of 10

```
      ENCODE (16,1,ALPHA)A,B
    1 FORMAT (A10,A6)
```

Result:



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.

```
      PROGRAM ENCO2(OUTPUT)
      I = 10RBCDEFGHIJK
      IA = 1H1
      ENCODE (8,10,I)I, IA, I
10    FORMAT (A3,A1,R4)
      PRINT 11,I
11    FORMAT (O20)
      END
```

Print-out is:

```
02030434101112135555
```

The display code equivalent is:

```
BCD1HIJKbb
```

DECODE

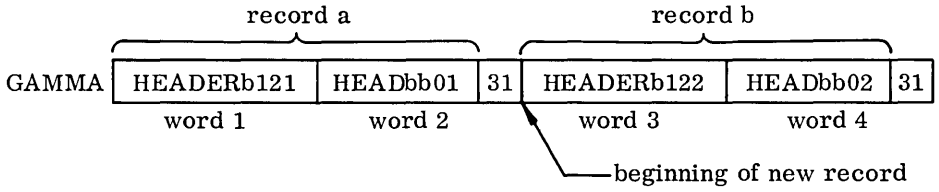
DECODE (n,f,A)k

The information in n consecutive BCD characters (starting at address A) is transmitted according to the FORMAT and stored in the list variables. If the record ends with a partial word the balance of the word is ignored. However, if the number of characters specified by the I/O list and the specification list f is greater than n (record length) per record, an execution diagnostic occurs. If DECODE attempts to process an illegal BCD code or a character illegal under a given conversion specification, that character is converted to a blank and conversion continues through n characters.

Examples:

1. n ≠ multiple of 10

```
      DECODE (18,1,GAMMA) A6,B6
1     FORMAT (A10,A8)
```



Result:

```
A6(1) = HEADERb121
A6(2) = HEADbb01bb
B6(1) = HEADERb122
B6(2) = HEADbb02bb
```

2. 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 BCD 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 BCD 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)
```

3. DECODE may be used to calculate a field definition in a FORMAT specification at object time. Assume that in the statement FORMAT (2A10, Im) the programmer wishes to specify m at some point in the program, subject to the restriction $2 \leq m \leq 9$. The following program permits m to vary.

```
      IF(M.LT.10.AND.M.GT.1)1,2
    1 ENCODE (10,100,SPECMAT)M
  100 FORMAT (7H(2A10,I,I1,1H))
          :
          :
      PRINT SPECMAT,A,B,J
```

M is tested to insure it is within limits. If not, control goes to statement 2 which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters: (2A10,I). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A10,Im).

A and B will be printed under specification A10, and the quantity J under specification I2, or I3, or ... or I9 according to the value of m.

SPECIFICATION AND DATA STATEMENTS 8

DIMENSION, COMMON, EQUIVALENCE, EXTERNAL, and TYPE statements, are called specification statements. Specification statements are nonexecutable statements which describe the characteristics, allocation and arrangement of data. The ordering of specification statements is immaterial, but they must appear before any statement function definition, DATA, NAMELIST, or executable statements in the program.

8.1 DIMENSION

Information necessary to allocate storage and define the reference for arrays may be provided by the DIMENSION statement.

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

Each v_j is a symbolic name and i_j is the corresponding subscript. Each i_j may consist of one, two, or three integer constants designating the dimensionality for the array and defining the maximum value which a subscript may assume in a subsequent array reference.

Example:

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

An array name may not contain a subscript which assumes a value during execution that is less than one or larger than the maximum length specified in the DIMENSION statement. If such a condition exists, an element beyond the array may be referenced. However, a subscript expression which assumes the value zero renders a result which is undefined.

The maximum value a subscript may attain is indicated below:

Dimensionality	Subscript Declarator	Subscript	Subscript Value	Maximum Subscript Value
1	(A)	(a)	a	A
2	(A, B)	(a, b)	a+A*(b-1)	A*B
3	(A, B, C)	(a, b, c)	a+A*(b-1) +A*B*(c-1)	A*B*C

a, b, c are subscript expressions.
A, B, C are dimensions.

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 issued.

Example:

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

The number of elements in the array CELL is 200. Since two words are used to contain a complex element, 400 words are reserved. 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, an informative diagnostic is issued and the first dimensions encountered are retained.

VARIABLE DIMENSIONS

If an entry in a declarator subscript is an integer variable name, the array is variable, and the variable names are called variable dimensions. Such an array may appear only in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer names that represent the variable dimensions. The values of the actual parameter list of the reference must be defined prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. Every array in an executable program requires at least one associated constant array declaration through subprogram references.

Example:

```
SUBROUTINE XMAX (DATA,K,J)
DIMENSION DATA (K,6,J)
```

In a subprogram, a symbolic name that appears in a COMMON statement may not identify a variable array.

DIMENSION statements must appear before any statement function definition, executable, DATA, or NAMELIST statements in the program.

8.2 COMMON

The COMMON statement reserves blocks of storage for variables or arrays appearing in one calling program or subprogram which may be shared and referenced with variables or arrays of other subprograms. The areas of common storage are specified by the statement form:

COMMON $/x_1/a_1/\ldots/x_n/a_n$

Each a is a non-empty set of variable names, array names or array declarators such as, v(i) illustrated for the DIMENSION statement, and each x is a block name. Block names may be symbolic names or integer constants in the range 0 to 9999999, but may not exceed seven characters in length.

Example:

```
COMMON/BLOCK1/A,T(10,15)/BLOCK2/E,G,Q
```

The list of variable names (A and T or E, G, and Q) may not be dummy parameters. The entries A and T are defined to be in the block labeled BLOCK1 and E, G, and Q are in the block labeled BLOCK2. These blocks are referred to as labeled common. However, if the block name is omitted as in:

```
COMMON/H/D,C,F//U,L,P(12,12)
```

or

```
COMMON S,V,Z,X,M
```

the list of variables following the empty block name specification are placed in unlabeled or blank common. In the two above examples, D, C, and F are in the block H, whereas U, L, P, S, V, Z, X, and M are defined in unlabeled common.

LABELED COMMON

Any labeled common block may be referred to by any number of programs or subprograms which comprise an executable program. References are made by block name which must be identical in all references. The definition of all labeled common blocks need not be made within any one program, but must be made in the program unit in which the data is needed.

The length of a common block in a program unit is the sum of the storage required for the elements defined by the COMMON statement. The length of labeled common blocks with the same label should be the same.

Example:

```
SUBROUTINE A              SUBROUTINE B
REAL B,W,X(20)            COMPLEX G,F(10)
COMMON/BLKA/V,W,X         COMMON/BLKA/G,F
```

Both references to the COMMON block, BLKA, correspond in size. That is, both subprograms define the block as containing 22 words; subroutine A specifies 22 items of real type and the specification in B indicates 11 items of complex type.

Reference may be made to the name of a labeled common block more than once in any program or subprogram. Multiple references may occur in a single COMMON statement, or the block name may be specified in any number of individual COMMON statements. In both cases, the processor links together all variables into a single labeled common block.

UNLABELED COMMON

All variables defined in unlabeled or blank common blocks are assigned together; that is, only one section of the storage allocated for common is assigned to such variables. These variables are always referred to by an unlabeled COMMON statement (block name is omitted).

Unlike labeled common, the sizes specified in various program units to be executed together need not be the same. Size is measured in terms of storage units.

Example:

```
SUBROUTINE ALPHA          SUBROUTINE BETA
COMMON E, F, G(20,10)     COMMON H,A,D,S
        :                         :
```

Subroutine ALPHA defines an area of 202 words in unlabeled common, BETA uses only 4 words or a maximum of 8 words of the storage already defined.

ARRANGEMENT OF COMMON BLOCKS

The properties of common block names as used in all of the program units of an executable program are as follows:

Each subprogram using a common block assigns the allocation of words in the block. The entities used within the block may differ as to name, type, and number of storage units although the block identifier itself must remain the same.

When a block is labeled and the entities are defined for the block, the values of identifiers in the corresponding positions (counted by the number of preceding storage units) are the values referenced through COMMON declaration in the executable program. The order of entities in the labeled common block is significant throughout the executable program.

Example:

```
PROGRAM MAIN (INPUT, OUTPUT)
      .
      .
      .
COMMON A1, A2, L1/B1/B2, B3
      .
      .
      .
CALL CALL1(S, T, Z)
      .
      .
      .
END
SUBROUTINE CALL1(X, Y, Z)
      .
      .
      .
COMMON A1, D, M/B1/F, G
      .
      .
      .
END
```

A double precision or a complex entity consists of two logical consecutive storage units: a logical, real, or integer entity is one storage unit.

If any common block elements are type ECS, all the elements of that block must be type ECS. No type ECS elements may appear in the blank common block.

COMMON statements must appear before any statement function definitions, executable, DATA, or NAMELIST statements in the program unit.

8.3 EQUIVALENCE

An EQUIVALENCE statement permits storage to be shared by two or more entities, it does not imply equality of entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor

EQUIVALENCE (k_1), (k_2), . . . , (k_n)

Each k is a list of the form:

$a_1, a_2, \ldots, a_m$

Each a is either a variable name or an array element name (but not a dummy argument or an ECS variable or array element), the subscripts may contain only constants. m is greater than or equal to two. The number of subscript expressions of an array element name must correspond to the dimensionality of the array declarator, or it must be one.

EQUIVALENCE may not be used to reorder COMMON nor reposition the base.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block beyond the last assignment for that block made by a COMMON statement.

When EQUIVALENCE is used for two variables or array elements, the name of the variables or arrays may not both appear in COMMON statements in the same program.

The following examples illustrate changes in block lengths as the result of EQUIVALENCE declaration.

Given: Arrays A and B

Sa subscript of A

Sb subscript of B

Examples:

1. A and C in common, B not in common

   Sb ≤ Sa is a permissible subscript arrangement

   Sb > Sa is not

	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)	

EQUIVALENCE statements must appear before any statement function definitions, executable, DATA, or NAMELIST statements in the program unit.

8.4 EXTERNAL

The EXTERNAL statement defines variable names to be external procedure names. This feature permits external procedure names to be passed as arguments to another external procedure; the names must be defined in an EXTERNAL statement in the program unit in which it is used.

EXTERNAL $v_1, v_2, \ldots, v_n$

v_i are declared to be external procedure names.

Example:

```
EXTERNAL NAME1,NAME2,NAME3
    :
    :
CALL SUB(A,B,NAME2)
SUBROUTINE SUB(X,Y,IFUNC)
```

The user is also allowed to define an Intrinsic function name in an EXTERNAL declaration. This re-definition of an intrinsic function name causes the processor to consider any subsequent reference as an external function reference; the user must supply the procedure.

EXTERNAL statements must appear before any statement function definitions, executable, DATA, or NAMELIST statements.

8.5 TYPE

The TYPE declaration provides the processor with information concerning the structure of variable and function identifiers. Six variable types may be declared by the statement:

$tv_1, v_2, \ldots, v_n$

t may be INTEGER, REAL, DOUBLE PRECISION (or DOUBLE), COMPLEX, LOGICAL or ECS optionally preceded by the characters TYPE. Each v is a variable name, array name, function name, or an array name with its dimensions which assumes the type indicated by t.

A TYPE statement may be used to override or confirm implicit typing; it must be used to declare entities to be double precision, complex, logical or ECS; it may also supply dimension information.

Example:

```
INTEGER ACBS, AFDS,ITRC
TYPE COMPLEX CC, F
```

The TYPE declaration is non-executable and must precede any statement function definitions, executable, DATA, or NAMELIST statements in a given program unit. Any variable defined by a TYPE statement may not be re-defined in another TYPE statement; when such a condition does exist, a diagnostic occurs and the processor assumes the type as declared when first encountered.

8.6 DATA

The DATA, data initialization, statement is used to define initial values of variable or array elements not located in blank COMMON.

DATA $k_1/d_1/, k_2/d_2/, \ldots, k_n/d_n/$

Each k is a list containing names of variables and/or array elements, but may not be dummy arguments. Each k may also be an array name which can have from one to three variable or integer constant control subscripts. Each d is a list of constants, optionally signed, which designate the values which each k is to assume.

Example:

```
DATA X, Y, Z/32.5, -7.4,3./,S, T/1.5E3,.TRUE./
```

Entries in the list are separated by commas. Hollerith constants may also be included.

The list d may be grouped by parentheses, any of which may be preceded by a repetition factor, j*.

Example:

```
DIMENSION AMASS(10,10,10), A(10), B(5)
DATA (AMASS(6,K,3),K=1,10)/4*(-2.,5.139),6.9,10./
DATA (A(I),I=5,7)/2*(4.1),5.0/
DATA B/5*0.0/
```

ARRAY AMASS:

AMASS(6,1,3) = -2.
AMASS(6,2,3) = 5.139
AMASS(6,3,3) = -2.
AMASS(6,4,3) = 5.139
AMASS(6,5,3) = -2.
AMASS(6,6,3) = 5.139
AMASS(6,7,3) = -2.
AMASS(6,8,3) = 5.139
AMASS(6,9,3) = 6.9
AMASS(6,10,3) = 10.

ARRAY A:

A(5) = 4.1
A(6) = 4.1
A(7) = 5.0

ARRAY B:

B(1) = 0.0
B(2) = 0.0
B(3) = 0.0
B(4) = 0.0
B(5) = 0.0

A one-to-one correspondence is necessary between the list items and the constants which establish their initial value.

Example:

```
DIMENSION K(10),A(2)
DATA A/2.0/
```

The value 2.0 is stored in A(1), however, in A(2), there is no definite value.

When the number of list elements exceeds the range of the implied DO, the excessive list elements are not stored.

Example:

```
DIMENSION B(10)
DATA(B(J),J=1,5)/4*1.23,6*1.24/
```

The excessive values 5*1.24 are discarded.

If a list item is an array name with no control subscripts or parameters, the constant list defines the values in the array to the maximum dimensional length or until the constant list is exhausted.

An initially defined variable or array element may not be in blank common.

An alternate form of the data initialization statement has the form:

DATA $(r_1=d_1)$, $(r_2=d_2)$, . . . , $(r_n=d_n)$

Each r is an array element name that may have from one to three control subscripts or a list of names of variables and array elements (each of which may be a single integer variable) and from one to three integer constant control parameters.

Each d is a list of constants and optionally signed constants, any of which may be preceded by j*. The constants may be grouped by parentheses and optionally preceded by j*; j is an integer constant.

Example:

```
DIMENSION D3(4), POQ(5, 5)
DATA (D3 = 5., 6., 7., 8.), (((POQ(I, J), I=1, 5), J=1, 5)=25*0)
```

which initializes:

D3(1) = 5.

D3(2) = 6.

D3(3) = 7.

D3(4) = 8.

and sets the entire POQ array to zero.

DATA statements must appear after all specification statements in a program unit.

The type of the DATA value is determined by the form of the constant, not the type of the list variables.

PROGRAM FUNCTION, SUBROUTINE, BLOCK DATA, AND LIBRARY ROUTINES

9

A FORTRAN Extended program consists of a main program with or without subprograms. Subprograms are separate programs that are executed only when called and may be defined by the programmer or be preprogrammed and contained in the processor or system library.

9.1 MAIN PROGRAM

The first statement of the main program must be one of the following forms; it may begin anywhere after column 6.

PROGRAM s

PROGRAM s (f_1,f_2,...,f_n)

s is a symbolic name of the main program, f_i are the names of all input/output files required by the main program and its subprograms.

The arguments must satisfy the following conditions within the program and its subprograms at compile time.

File name INPUT must appear if the READ f,k statement is included.

File name OUTPUT must appear if any PRINT statement is included: also needed for printing of execution time diagnostics.

File name PUNCH must appear if any PUNCH statement is included.

File name TAPE i (i is an integer constant 1-99) must appear if any input/output statement involving unit i appears in the program. If i is a variable, there must be a file name TAPE i for each value i may assume.

Files may be equivalenced at compile time. For example,

(INPUT, OUTPUT, TAPE1 = INPUT, TAPE2=OUTPUT)

All input normally provided by TAPE1 is to be extracted from INPUT and all listable output normally recorded on TAPE2 is to be transmitted to the OUTPUT file.

In the list of parameters, equivalenced file names must follow those to which they are made equivalent.

File buffers may be assigned a non-standard size at compile time; (OUTPUT=4000,TAPE4=OUTPUT). If buffer size is not indicated, 1025 is assumed. If the buffer is explicitly assigned a length, the assignment must appear with the first reference to the file on the program card. The length may be specified in decimal or in octal with the trailing B.

If the PROGRAM card is omitted, the FORTRAN processor assumes a program name of START. when it encounters a statement that is not a comment card. Input/output buffers and files for the program are equated to the standard SCOPE system files INPUT and OUTPUT.

The equivalencing of files causes associated buffer and file names to be equivalenced.

Example:

```
PROGRAM HELLO (TAPE1,TAPE2=TAPE1)
    .
    .
    .
N=1
WRITE (N) A
    .
    .
    .
END

PROGRAM HELLO (TAPE1,TAPE2=TAPE1)
    .
    .
    .
N=2
WRITE(N) A
    .
    .
    .
END
```

The file name resulting from both programs is TAPE 1.

The file names declared on the program card are the only names that may result from I/O references within the program. If no parameters appear on the control card which calls a program into execution, the non-equivalenced declared names will be taken as the SCOPE file names to be accessed. If parameters do appear on the control card which calls a program into execution, each parameter will be the SCOPE file name to be accessed by the corresponding program declared name. In a program headed by the program card

PROGRAM name $(f_1,f_2,\ldots,f_n)$

which is called into execution by the control card

LGO$(p_1,p_2,\ldots,p_n)$

(where each p_i may be null), a reference to the declared name f_i will access the SCOPE file f_i if p_i is null; otherwise, the SCOPE file p_i will be accessed. Only non-equivalenced program declared names may have a corresponding p_i specified on the control card which calls the program into execution.

Example:

If a program is headed by the card

PROGRAM PROG (TAPE1,OUTPUT,TAPE2=OUTPUT)

and is called into execution with

LGO.

every reference to unit 1 within PROG will access the SCOPE file TAPE1, every print statement and every reference to unit 2 will access the SCOPE file OUTPUT.

If PROG is called into execution with the control card

LGO(INPUT, LOAD)

every reference to unit 1 within PROG will access the SCOPE file INPUT; every refer print statement or reference to unit 2 will access the SCOPE file LOAD.

Calling PROG into execution with the control card

LGO(, , LOAD)

will act the same as using

LGO.

in the former case, there is an illegal attempt to change an equivalenced declared name (the attempt is ignored).

9.2 SUBROUTINE SUBPROGRAMS

A subroutine is an external computational procedure defined by FORTRAN statements which is identified by a SUBROUTINE statement and may or may not return values to the calling program. The statement may have any one of the following forms:

SUBROUTINE s $(a_1,a_2,\ldots,a_n)$ or SUBROUTINE s
SUBROUTINE s $(a_1,a_2,\ldots,a_n)$, RETURNS $(b_1,b_2,\ldots,b_m)$

or

SUBROUTINE s,RETURNS $(b_1,b_2,\ldots,b_m)$

s is the symbolic name of the subroutine, a_i are the dummy arguments (these may be variable names, array names or external procedure names), and b_i are variable names containing statement labels which indicate alternate exits from the subroutine. SUBROUTINEs and FUNCTIONs are restricted to a maximum of 63 dummy arguments.

Example:

```
      Calling Program                    Subprogram
           .
           .
           .                        SUBROUTINE PGM1 (X, Y, Z),
      CALL PGM1 (A, B, C),            RETURNS (M, N)
        RETURNS (5, 10)               U=V*W+T**2
           .                          X=Y*Z
           .                     20   IF (U+X) 25, 30, 35
           .                     25   RETURN M
 5    B=SQRT(A*C)                30   RETURN N
           .                     35   Z=Z+(X*Y)
           .                          RETURN
10    CALL PGM2 (D, E)                END
           .
           .
           .
```

The above example illustrates the different types of returns which may be made from a subroutine subprogram. If the RETURNS list is omitted from the CALL statement in the calling program, a return of the form RETURN a may not be made. However, the converse is permitted; a normal return via the RETURN statement may be made to the calling program if the RETURNS list is specified in the CALL statement.

Subroutine subprograms are constructed with the following restrictions:

Symbolic name of the subroutine must not appear in any other statement in this subprogram.

Symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

Subroutine subprograms do not require a RETURN statement if the procedure is completed upon executing the END statement. When the end line is encountered, a return is implied.

Subroutine subprograms may contain any statements except BLOCK DATA, FUNCTION, or another SUBROUTINE statement.

Execution of a subroutine begins with the first executable statement of the subprogram. Continuation is sequential unless a GO TO, IF, RETURN, STOP or terminal statement of a DO is encountered, in which case execution proceeds as indicated.

A reference to a subroutine is made by a CALL statement. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program; otherwise the results are unpredictable. The use of a Hollerith constant or octal constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

Constant

Variable name

Array element name

Array name

Name of an external procedure

ECS variable name

ECS array element name

ECS array name

Any other expression

Several restrictions and rules govern the correspondence of actual arguments in the calling program to dummy arguments in the subprogram:

If an argument in the calling program is an external function or subroutine name, the corresponding dummy argument must be used in the same manner.

An argument in the calling program must be a variable name, an array element name, or an array name if it corresponds to a dummy argument which is defined or redefined in the subprogram.

The association of arguments in the calling program is made by name to dummy arguments appearing in executable statements, function definition statements, or those used as adjustable dimensions in the subprogram. However, if an argument takes the form of an expression (any other expression), the association is by value rather than by name.

An argument which is an array element name containing variables in the subscript expression may be replaced by the same argument with a constant subscript with an equivalent value.

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Example:

Assume X = 3 and Y = 2

1)
```
CALL SUBA (X,X)          SUBROUTINE SUBA (A, B)
                              .
                              .
                              .
                         A = Y
                         Z = B
```

2)
```
COMMON X                 SUBROUTINE SUBB (B)
CALL SUBB (X)            COMMON A
     .                        .
     .                        .
     .                        .
END                      A = Y
                         Z = B
                         END
```

In the above examples, the first two statements in the subroutine set X = Y then Z = X resulting with X = 2 and Z = 2. However, if the statements are reversed the results obtained will be different; Z = X then X = Y, the numeric values resulting are Z = 3 and X = 2.

ENTRY STATEMENT

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

```
ENTRY name
```

Name is an alphanumeric identifier which may appear within the subprogram only in the ENTRY statement. Each entry identifier must appear in a separate ENTRY statement. The formal parameters, if any, appearing with the FUNCTION or SUBROUTINE statement do not appear with the ENTRY statement. They are assumed to be the same as those of the FUNCTION or SUBROUTINE in which the ENTRY statement is located. ENTRY may appear anywhere within the subprogram except within a DO; ENTRY statements 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 if reference were being made to the function or subroutine in which the ENTRY is imbedded. The name may appear in an EXTERNAL statement and, if a function entry name, in a TYPE statement.

The ENTRY name type must agree with the function name type. The name may not be given a 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 JAM
      IF (X.GT.Y) 10,20
20    JOE=X-Y
      RETURN
      END
```

This could be called from the main program as follows:

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

LIBRARY SUBROUTINES

Library subroutine subprograms may be referred to by any program with a CALL statement. i must be an integer variable or constant, j is an integer variable.

CALL SLITE (i) Turn on sense light i. If i = 0, turn all sense lights off. i is 0 to 6; if i > 6 or < 0, an informative diagnostic is given and all sense lights remain unchanged.

CALL SLITET (i,j) If sense light i is on, j = 1, if sense light i is off, j= 2; then turn sense light i off. i is 1 to 6. If i > 6 or < 0, an informative diagnostic is given and all sense lights remain unchanged and j=2.

CALL SSWITCH (i,j) If sense switch i is on (down), j = 1, if sense switch i is off (up), j = 2, i is 1 to 6. If i > 6 or < 0, an informative diagnostic is given and all sense switches remain unchanged and j = 2.

CALL EXIT Terminate program execution and return control to the operating system.

CALL REMARK (H) Place a message, not to exceed 40 characters, in the dayfile. H is a Hollerith specification.

CALL DISPLA(H,k) Displays a variable name and its numerical value in the dayfile. The value k is displayed as an integer if not normalized and in floating point format if normalized. H is a Hollerith specification.

CALL RANGET(n) Obtain current generative value of RANF between 0 and 1. n is a symbolic name.

CALL RANSET(n) Initialize generative value of RANF. n is real.

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

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

Dump storage on the OUTPUT file in the indicated format. If PDUMP was called, return control to the calling program; if DUMP was called, terminate program execution and return control to the monitor. a_i and b_i identifiers indicate the first word and the last word of the storage area to be dumped; $1 \leq n \leq 20$. The dump format indicators are as follows:

f = 0 or 3 octal dump

f = 1 real dump

f = 2 integer dump

f = 4 octal dump; this implies that a_i and b_i are statement numbers that have been defined by an ASSIGN statement.

9.3 FUNCTION SUBPROGRAMS

STATEMENT FUNCTIONS

Statement function definitions must precede the first executable statement of the program or subprogram and must follow any specification statements. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program or subprogram. A statement function applies only to the program or subprogram containing the definition; it is defined by a statement of the form:

$f(a_1,a_2,\ldots,a_n) = e$

f is the statement function name, e is any expression. a_i are variable names which are dummy arguments indicating type, number, and order of arguments; they may be the same as variable names of the same type appearing elsewhere in the program unit. n may not exceed 63. f and e must be both logical or both non-logical.

Examples:

1. LOGICAL C, P, EQV
   EQV(C, P) = (C. A. P). O. (. N. C. A. . N. P)
2. COMPLEX D, F
   D(A, B) = (3.2, 0.9)*EXP(A)*SIN(B)+(2.0, 1.)*EXP(Y)*COS(B)
3. GROS(R, HRS, OTHERS) = R*HRS + R* .5*OTHERS

INTRINSIC FUNCTION

The symbolic names of the intrinsic functions (built-in functions) have special meaning and type as described in Appendix D. An intrinsic function may be referenced when it is used as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Appendix D and may be any expression of the specified type.

Examples:

1) DATA(I)=DATA(I+1)*((FLOAT(MAX)/K(I))/DATA(I))

2) IB(J)=IFIX(B(J))

The intrinsic functions SIGN, ISIGN, and DSIGN are defined when the value of the second argument is zero, such that the sign of the second argument is taken as positive (negative) for +0(-0). However, the functions AMOD and MOD are not defined when the second argument is zero; division by zero renders the results undefined.

EXTERNAL FUNCTION

An external function is defined externally to the program or subprogram that references it. An external procedure defined by FORTRAN statements headed by a FUNCTION statement is called a function subprogram.

t FUNCTION $f(a_1,a_2,\ldots,a_n)$ or FUNCTION $f(a_1,a_2,\ldots,a_n)$

t is INTEGER, REAL, DOUBLE, DOUBLE PRECISION, COMPLEX, LOGICAL, or it is omitted.

f is the symbolic name of the function. If t is omitted the type of the function is derived from f according to the type rules of implicit definition.

a_i are the dummy arguments; each may be a variable name, an array name, or an external procedure name. $1 \leq i \leq 63$.

The function name f must appear as a variable in the defining subprogram. During every execution of the subprogram, the variable must be defined, and once defined, it may be referenced or redefined. The value of the variable when a RETURN statement is executed is the value of the function. The function name f must not appear in any non-executable statement other than the FUNCTION statement in the function subprogram.

The dummy argument names may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram. The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

A function subprogram may contain any statement except BLOCK DATA, SUBROUTINE, another FUNCTION STATEMENT, or any statement that directly or indirectly references the function being defined.

When the END line is reached, a return is implied.

Example:

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

EXTERNAL FUNCTION REFERENCE

The reference to an external function may also be established when it is used as an operand in an arithmetic or logical expression. The actual arguments must agree in order, number, and type with the corresponding dummy arguments in the defining program.

$f(a_1,a_2,\ldots,a_n)$

$\underline{f}$ is a symbolic name of the function, $\underline{a}_i$ are the actual arguments. An actual argument name in an external function reference may be one of the following:

Variable

Array element

Array name

External procedure reference

Constant

ECS variable

ECS array

ECS array element

Any other expression

The rules governing the association of arguments in the function call to dummy arguments in the function are the same as those enumerated for subroutine subprograms.

Examples:

1) W(I,J)=FA+FB-GRATER(C-D,3.*AX/BX)

2) FUNCTION PHI (ALPHA, PHI2)
   PHI=PHI2(ALPHA)
   RETURN
   END

The reference to the function PHI in example 2 may be:

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

The replacement statement in the function PHI will produce the same result as if it had been written PHI=SIN(Q(K)).

BASIC EXTERNAL FUNCTIONS

The basic external functions listed in Appendix D are referred to in the manner described in the section, External Function Reference. Arguments may not be used for which a result is not mathematically defined and they may not be of a type other than that specified.

9.4 BLOCK DATA SUBPROGRAM

Initialization of data to be stored in labeled common may be accomplished by the specification of a BLOCK DATA subprogram which begins with a statement of the form:

```
BLOCK DATA
```

or

```
BLOCK DATA d
```

d is the symbolic name of the BLOCK DATA subprogram. This parameter must be specified if the subprogram is to be included in a SEGMENT as defined for SCOPE.

The BLOCK DATA subprogram contains only specification and DATA statements; executable statements are prohibited. Only the DATA, COMMON, DIMENSION, EQUIVALENCE, and TYPE statements associated with the data being defined are accepted; data may not be entered into an unlabeled (blank) common block. If an entry for a common block is given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements.

Example:

```
BLOCK DATA
COMMON/MAX/DATA(3),AA,BB/MIN/A,B,C,LAX
REAL LAX
INTEGER BB
COMPLEX A
DOUBLE PRECISION C
DATA LAX/145.12/,(DATA(I),I=1,3)/1.1,2*9.3/,BB/1256/,A,B,C/
    (2.0,1.0),13.6,172.5432D06/
END
```

Initial values may be entered into more than one block in a single subprogram.

OVERLAYS AND SEGMENTS

Programs that exceed available memory may be divided into independent parts which may be called and executed as needed. Such programs can be divided into segments or overlays.

Segments are groups of subprograms that are loaded in relocatable form when requested, giving the user explicit control over established interprogram links. An overlay is a program combined with its subprograms which is converted to absolute form and written on mass storage prior to execution. During execution, overlays are called into memory and executed as requested. OVERLAY and SEGMENT loader control cards are recognized by the compiler if they start in column 7 or later and appear between subprograms. Compiler processing places them in the desired position on the binary output file.

10.1 OVERLAYS

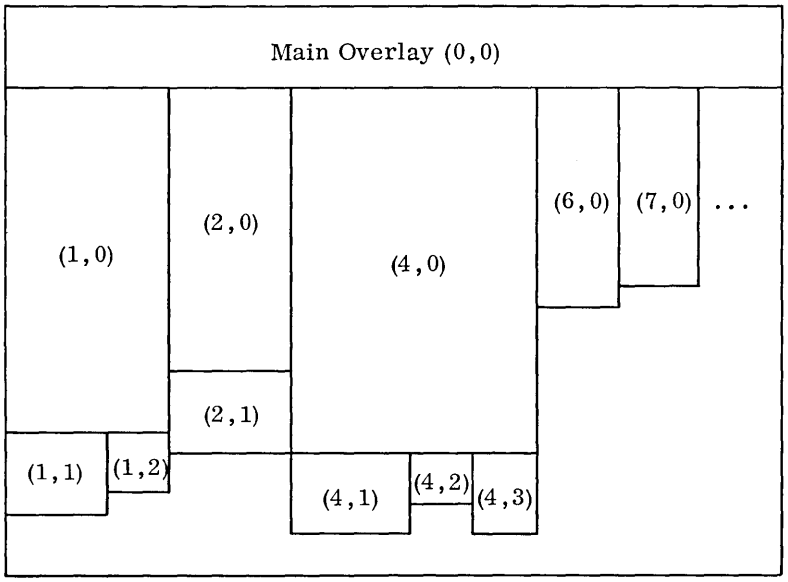
Overlay processing allows programs to be divided into independent parts which may be called and executed as needed. Each part (overlay) must consist of a single main program and any necessary subprograms.

Each overlay is numbered with an ordered pair of numbers (I, J), each in the range $0\text{-}77_8$. I denotes the primary level, J, the secondary level. An overlay with a non-zero secondary level is called a secondary overlay and is associated with and subordinate to the overlay which has the same primary leve and a zero secondary level, called the primary overlay. The initial or main overlay which always remains in memory has levels (0, 0). The significance of this distinction appears in the order in which overlays are loaded.

Overlay level numbers, (0, 1), (0, 2), (0, 3)...are illegal. Primary overlays all have their origin at the same point immediately following the main overlay (0, 0). The origin of secondary overlays immediately follows the primary overlay. For any given program execution, all overlay identifiers must be unique. The loading of any primary overlay destroys any other primary overlay. For this reason, no primary overlay may load other primary overlays. Secondary overlays may be loaded only by the associated primary overlay or main overlay. Thus two levels of overlays are available to the programmer.

An overlay may reference subprograms in its own overlay, in the main overlay, or in its associated primary overlay.

Example:



Overlays (1,1) and (1,2) are secondary to overlay (1,0)

Overlay (2,1) is secondary to overlay (2,0)

Overlay (2,1) may not be called from (1,0) or (1,1) or (1,2) but only from (2,0) or (0,0)

Overlays (1,0), (2,0), (4,0)...may be called only from the main overlay (0,0)

OVERLAY CONTROL CARDS

OVERLAY (lfn, l1, l2, Cnnnnnn)

lfn — File name on which overlay is to be written; first overlay card must have a named lfn. Subsequent cards may omit it, indicating that the overlays are related and are to be written in the same lfn. A different lfn on subsequent cards results in generation of overlays to the new lfn.

l1 — Primary level number in octal.

l2 — Secondary level number in octal. l1, l2 for the first overlay card must be 0,0.

Cnnnnnn — Optional parameter consisting of letter C and six-digit octal number. If this parameter is present, the overlay is loaded nnnnnn words from the start of blank common. This provides a method of changing the size of blank common at execution time. Cnnnnnn cannot be included on the overlay 0,0 loader directive. If this parameter is omitted, the overlay is loaded in the normal manner.

Overlays are called by:

CALL OVERLAY (fn, I, J, p, l)

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

- fn Variable name of a location which contains the name of the file (left justified display code) that contains the overlay
- I Primary level of overlay
- J Secondary level of overlay
- p Recall parameter. If p equals 6HRECALL, the overlay is not reloaded if it is already in memory.
- l Load parameter. Used to determine which value of the fn will be used. l may be any value. If l is present and non-zero, the overlay designated by fn will be loaded from the system library; otherwise, it will be loaded from the file designated by fn.

Prior to execution of this call which causes loading and execution of the overlay, the overlay must have been made absolute and written on file fn. When an END statement in the main program of an overlay is encountered, control returns to the statement following the CALL OVERLAY which initialized execution of the overlay in question.

10.2 SEGMENTS

A segment is a group of subprograms (possibly one) which are loaded together when specified by the programmer. Segments are loaded at levels from $0\text{-}77_8$. Level zero is reserved for the initial or main segment. Level zero, which must contain a PROGRAM, remains in memory during execution.

The following definitions apply to segmentation.

Entry point. A named location within a subprogram that can be referenced by another program — created by the SUBROUTINE, FUNCTION and ENTRY statements.

External reference. A reference within a program or subprogram to the entry point of some other subprogram — created by explicit CALL statements, function references, I/O statements, implicit functions, etc.

Link. The connection established between an external reference and an entry point when the programs are loaded into memory.

Unsatisfied external. An external reference for which no matching entry point can be found, and therefore no link established.

When the segment is loaded, external references will be linked to entry points in previously loaded segments (those at a lower level). Similarly, entry points in the segment are linked to unsatisfied external references in previously loaded segments. Unsatisfied external references in the segment remain unsatisfied; subsequent segment loading may include entry points to satisfy the external references. Unsatisfied external references may be satisfied, if possible, from the system library.

If a segment is to be loaded at a requested level which is less than or equal to the level of the last loaded segment, all segments at levels down to and including the requested level will be removed/delinked. Delinking a segment at a given level requires that the linkage of external references in lower levels to entry points in the delinked segment be destroyed so that the external references are unsatisfied once again.

Once the delinking is complete, the segment is loaded. Only one occurrence of a given subprogram or entry point is necessary since all levels may eventually link to the subprogram. However, a user may force loading of a subprogram by explicitly naming it in another segment at a higher level. Thereafter, all external references in higher levels are linked to the new version. In this manner, a subprogram and/or entry point can effectively replace an identical one already loaded at a lower level. However, once a linkage is established, it is not destroyed unless the segment containing the entry point is removed.

Example:

> The SINE routine is loaded in a segment at level 1. The user wishes to try an experimental version of SINE. He loads a segment containing the new SINE at level 2. Segments loaded at level 3 or higher will now be linked to SINE at level 2 until a new level 2 or a new SINE is loaded.

Common blocks may be loaded with any segment. Labeled common may not be cross-referenced in segments. Maximum blank common length is established in the first segment which declares blank common.

10.2.1 SEGMENT CONTROL CARDS

SECTIONS

This card defines a section within a segment. Segments are loaded by user calls during execution or by MTR during initial load.

SECTION (sname, pn_1, pn_2, . . . , pn_n)

sname	Name of section (7-characters maximum).
pn_i	Names of subprograms in the section. If more than one card is necessary to define a section, additional cards with the same sname may follow consecutively.

All subprograms within a section are loaded whenever the named section is loaded. All section cards must appear prior to the SEGMENT cards which refer to the named sections.

SEGMENTS

All programs requiring segmentation loading must contain a SEGZERO card prior to any of the binary text.

SEGZERO (sn, pn_1, pn_2, . . . , pn_n)

- sn — Segment name
- pn_i — Names of subprograms or section names which make up main or zero level segment. Defining other segments in a similar manner reduces the list of subprograms in the loader call.

SEGMENT (sn, pn_1, pn_2, . . . , pn_n)

The parameters are defined as in SEGZERO. In a segment, all programs must reside on the same file. A segment defined in the user's program need not be defined by a SEGMENT card; however, a SEGZERO card is always required.

Segments may be loaded by;

CALL SEGMENT (fn, e, a, lib, m)

- fn — Variable name of location which contains the file name (left justified display code) from which the segment load takes place.
- e — Level of the segment load.
- a — Variable name of array containing a list of SEGMENTS, SECTIONS and/or SUBPROGRAMS to be loaded with this call. In this list, the name must be in left justified display code, and the list must be terminated by a zero entry. An initial list entry of zero signals a segment load of all subprograms remaining on the file fn.
- lib — If zero or blank, unsatisfied externals are to be satisfied, if possible, from the system library.
- m — If zero or blank, a map of the segment load is not produced. lib and m need not be specified.

Once the named subprograms are loaded control returns to the statement following the CALL SEGMENT. The programmer is free to call on the loaded subprograms as desired.

DEBUGGING FACILITY 11

The debugging mode of compilation, along with the source cross-reference map selection, is provided specifically to aid in the development or conversion of programs. In the debugging mode of compilation, a programmer can establish a record of selected operations as they are performed in the execution of his program. This mode facilitates debugging from a source listing, and perhaps a source cross-reference map should core dumps be required; it makes their interpretation much easier.

Features provided with the debugging mode of compilation:

- Array bounds checking
- Program flow tracing
- Call and return tracing
- Function call and value returned tracing
- Stores checking
- Assigned GO TO checking
- Partial execution of routines containing fatal errors

The debugging mode is selected by the option D on the FTN control card (Appendix C). In this mode, debugging selection cards are recognized. If this mode is not specified, debugging selection cards are treated as comments.

In the debugging mode, a program is compiled so that specified checks can be performed during execution; however, execution will stop when a fatal error is detected.

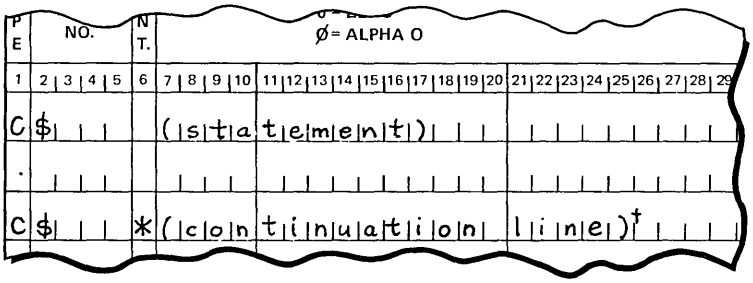
When a program is compiled in debug mode, 12000_8 words will be required beyond the minimum field length for non-debug mode compilation. To execute, 2500_8 words beyond the minimum will be required.

11.1 FORMAT

Debugging statements are punched in columns 7-72, as in the normal FORTRAN statement. In addition:

Columns 1 and 2 of each statement must contain the characters C$

A continuation line must be flagged by a character in column 6 (any FORTRAN character other than blank or zero). Columns 3-5 must be blank.



† if required

The restriction on the number of debug continuation lines is the same as for FORTRAN continuation lines. When FORTRAN Extended is not in debug mode or when the program is used with another FORTRAN compiler, the debug cards will be treated as comment cards. Since even working programs sometimes exhibit new bugs, it could be advantageous to retain the debugging statements in a program once checkout is complete.

In the following pages, excerpts from an actual printout of a working program are used in conjunction with typewritten examples to illustrate the debugging messages. A sample working program is reproduced in full at the end of the chapter.

11.2 ARRAYS STATEMENT

The ARRAYS statement initiates subscript bounds checking on specified arrays. Warning messages appear on the output if the address calculated by the array indexing function is not within the storage allocated for the array.

C$ ARRAYS($a_1, a_2 \ldots a_n$)

C$ ARRAYS

($a_1 \ldots a_n$) are the names of the arrays for which subscript bounds are to be checked. If array names are not given, all arrays in the program unit are checked.

The C$ ARRAYS statement does not provide checking of individual subscripts, only checking of the address computed from all the subscripts.

When ARRAYS statement is used, a bounds check is made each time an element of an array is referenced. Bounds checking is not performed for array references in an input/output list. If the element is not within the overall bounds of the array, a message is printed with the job output, as shown in the following example. After printing a message for an out of bounds array reference, the reference is allowed to occur.

```
/DEBUG/  SAMPLE  AT LINE   11- THE SUBSCRIPT VALUE OF     6 IN ARRAY A1   EXCEEDS DIMENSIONED BOUND OF   5
/DEBUG/  SAMPLE  AT LINE   11- THE SUBSCRIPT VALUE OF     0 IN ARRAY A1   EXCEEDS DIMENSIONED BOUND OF   5
/DEBUG/  SAMPLE  AT LINE   13- THE SUBSCRIPT VALUE OF     6 IN ARRAY A1   EXCEEDS DIMENSIONED BOUND OF   5
/DEBUG/  SAMPLE  AT LINE   14- THE SUBSCRIPT VALUE OF     0 IN ARRAY A1   EXCEEDS DIMENSIONED BOUND OF   5
```

Identifies a debugging message	Program unit name containing subscript reference	Line number of reference	Value of subscript in reference	Name of array being referenced	Actual dimension limits of array

11.3 CALLS STATEMENT

This statement traces calls to and returns from specified subroutines.

C$ CALLS($a_1, \ldots, a_n$)

C$ CALLS

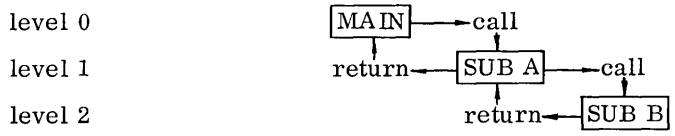
The subroutine names for which call tracing is to be performed are indicated by($a_1 \ldots a_n$). If this parameter is not specified, all subroutine calls are traced. Nonstandard returns are also traced.

The message produced for each call and return are printed with the job output as follows:

```
/DEBUG/  SAMPLE  AT LINE   23- ROUTINE SUB1   CALLED  AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   24- ROUTINE SUB1   RETURNS TO LEVEL   0
/DEBUG/  SAMPLE  AT LINE   25- ROUTINE SLITE  CALLED  AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   26- ROUTINE SLITE  RETURNS TO LEVEL   0
```

Identifies a debugging message	Program unit name containing reference	Line number containing call or return	Name of subroutine called or returned	Indicates call or return status and level number

A main program is always at level zero; subroutines are at any level other than zero. Calls are always in order of ascending level number; returns are always in order of descending level number.



Traceback information from the current subroutine level back to the main level is available through a call STRACE, an entry point in the object routine BUGCLL. The output is printed on a file named DEBUG; however, the program need not be compiled in debug mode to use this feature.

```
PROGRAM      MAIN

                 PROGRAM MAIN (OUTPUT,DEBUG=OUTPUT)
                 CALL SUB1
                 END

SUBROUTINE   SUB1

                 SUBROUTINE SUB1
                 CALL SUB2
                 RETURN
                 END

SUBROUTINE   SUB2

                 SUBROUTINE SUB2
                 I = FUNC1(2)
                 RETURN
                 END

FUNCTION     FUNC1

                 FUNCTION FUNC1 (K)
                 FUNC1 = K ** 10
                 CALL STRACE
                 RETURN
05               END
```

Output from STRACE:

```
/DEBUG/ FUNC1  AT LINE  3- TRACE ROUTINE CALLED
                        FUNC1  CALLED BY SUB2  AT LINE  2, FROM  1 LEVELS BACK
                        SUB2   CALLED BY SUB1  AT LINE  2, FROM  2 LEVELS BACK
                        SUB1   CALLED BY MAIN  AT LINE  2, FROM  3 LEVELS BACK
```

11.4 FUNCS STATEMENT

Function tracing is similar to call tracing except that functions return a value that often is of concern to the programmer.

C$ FUNCS(a_1,...,a_n)

C$ FUNCS

The function names for which function tracing is to be performed are indicated by (a_1,...,a_n). If no names are listed, all functions are traced. Functions used in array subscripts in input/output lists and statement functions are not traced. A message is issued for each use of a function; it is printed with the job output as shown below.

```
/DEBUG/  SAMPLE  AT LINE   33- REAL    FUNCTION FUN1   CALLED AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   33- REAL    FUNCTION FUN1   RETURNS A VALUE OF   7743.000000     AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   35- INTEGER FUNCTION IABS   CALLED AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   35- INTEGER FUNCTION IABS   RETURNS A VALUE OF          8242 AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   37- REAL    FUNCTION EXP    CALLED AT LEVEL   0
/DEBUG/  SAMPLE  AT LINE   37- REAL    FUNCTION EXP    RETURNS A VALUE OF   23114063123     AT LEVEL   0
```

Identifies a debugging message	Program unit using functions	Line number containing the function usage	Function type	Function name	Level number of using program unit including call or return status	Value returned by function	Level to which value is being returned

11.5 STORES STATEMENT

The STORES statement is used to record changes in value of specified variables resulting from arithmetic assignment statements. Variables altered as a result of use in an input list or a subroutine (function) parameter list are not detected. Stores checking is not performed on the control variable of a DO loop; stores checking is not performed when a variable is changed as a result of a store into an equivalenced variable.

C$ STORES($c_1, c_2, \ldots, c_n$)

($c_1, \ldots c_n$) can be variable names or relational expressions in the form:

variable name .relational operator. constant

or expressions with checking operators in the form:

variable name .checking operator.

The checking operators are:

RANGE	prints when the value is out of range
INDEF	prints when the value is indefinite
VALUE	prints for either out of range or indefinite

If variable names are used, a message is issued each time a new value is stored in a variable or array element. If the relational or checking expression is used, a message is issued only when the stored value satisfies the relation. The message will contain:

```
/DEBUG/  SAMPLE  AT LINE   48- THE NEW VALUE OF THE VARIABLE A1      IS   1.000000000
/DEBUG/  SAMPLE  AT LINE   48- THE NEW VALUE OF THE VARIABLE A1      IS   2.000000000
/DEBUG/  SAMPLE  AT LINE   48- THE NEW VALUE OF THE VARIABLE A1      IS   3.000000000
/DEBUG/  SAMPLE  AT LINE   48- THE NEW VALUE OF THE VARIABLE A1      IS   4.000000000
/DEBUG/  SAMPLE  AT LINE   48- THE NEW VALUE OF THE VARIABLE A1      IS   5.000000000
/DEBUG/  SAMPLE  AT LINE   51- THE NEW VALUE OF THE VARIABLE AGAIN   IS   3.141590000
/DEBUG/  SAMPLE  AT LINE   53- THE NEW VALUE OF THE VARIABLE A2      IS   5.000000000
/DEBUG/  SAMPLE  AT LINE   54- THE NEW VALUE OF THE VARIABLE IAGAIN  IS             10
/DEBUG/  SAMPLE  AT LINE   54- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              9
/DEBUG/  SAMPLE  AT LINE   54- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              8
/DEBUG/  SAMPLE  AT LINE   54- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              7
/DEBUG/  SAMPLE  AT LINE   54- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              6
```

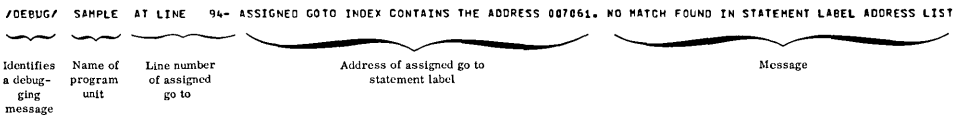
Identifies a debugging message	Name of program unit	Line number of reference	Name of variable, and message	New value of variable

11.6 GOTOS STATEMENT

This statement checks the validity of the selected statement labels in an in an assigned GO TO.

```
C$    GOTOS
```

The statement label assigned to the integer variable is compared with statement labels in the list. A message is printed when the label value is not in the list, but the transfer of control is allowed to occur.



11.7 TRACE STATEMENT

When the TRACE statement is used, a message is produced for each intra-program transfer of control at a level less than or equal to the level specified by lv.

```
C$    TRACE(lv)

C$    TRACE
```

If lv = 0, tracing will occur only outside DO loops; if lv = n, tracing will occur up to and including level n in a DO nest; if no level is specified, zero level is implied. If a DO loop is not satisfied, the transfer back to the start of the loop is not traced. Transfers resulting from nonstandard returns are not traced. (These may be checked using C$ CALLS.) When tracing is selected and an out-of-bound computed GO TO is executed, the value of the incorrect index is printed before the job is terminated.

Flow tracing will follow these types of program flow control:

Simple GO TO
Computed GO TO
Assigned GO TO
Arithmetic IF
True side of logical IF

The output message will contain the following:

```
/DEBUG/  SAMPLE  AT LINE   71- CONTROL WILL BE TRANSFERRED TO STATEMENT 503   AT LINE    73
/DEBUG/  SAMPLE  AT LINE   73- CONTROL WILL BE TRANSFERRED TO STATEMENT 504   AT LINE    75
/DEBUG/  SAMPLE  AT LINE   75- CONTROL WILL BE TRANSFERRED TO STATEMENT 505   AT LINE    77
/DEBUG/  SAMPLE  AT LINE   77- CONTROL WILL BE TRANSFERRED TO STATEMENT 506   AT LINE    78
```

Identifies a debugging message

Program unit name

Line number from which control transferred

Statement number to which control was transferred

Line number of statement to which control was transferred

11.1 NOGO STATEMENT

The NOGO statement suppresses partial execution of a compiled routine whenever a fatal compilation error occurs during compilation.

```
C$    NOGO
```

If the NOGO statement is not present and the debugging mode is in effect, the program executes until a fatal error is encountered; at which point, the following message is issued:

```
FATAL ERROR ENCOUNTERED DURING PROGRAM
EXECUTION DUE TO COMPILATION ERROR.
```

Partial execution is not permitted for only three classes of errors:

Errors in the declarative statements

Missing DO loop terminators

Missing FORMAT statement numbers

11.9 DECK STRUCTURE

Debugging statements may be interspersed with FORTRAN statements in the source deck of a program unit (main program, subroutine, function). The debugging statements apply to the program unit in which they appear. Inclusion of interspersed debugging statements will change the FORTRAN generated line numbers for a program (figure 11-1).

Debugging statements also may be grouped to form a debugging deck beginning with a C$ DEBUG card. Debugging decks may be placed in a job in one of the following ways:

As on external debugging deck in a separate file named by the D parameter on the FTN card. When no name is specified by the D parameter, the INPUT file is assumed. (Figure 11-2.)

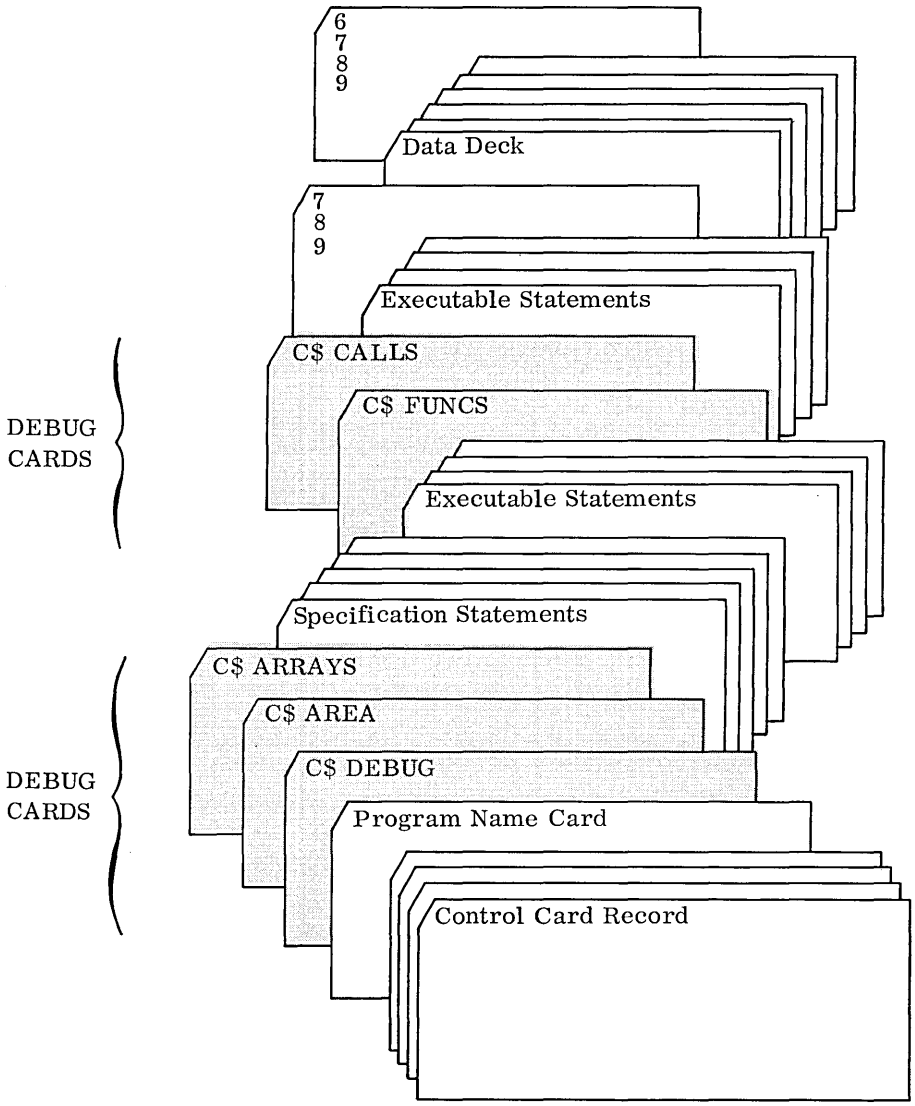
Immediately preceding the first source deck in the compiler input record (External Packet, figure 11-3).

Immediately after a program header card (PROGRAM, SUBROUTINE, or FUNCTION statement) (Internal Packet, figure 11-4).

The range of a debugging statement depends on its position:

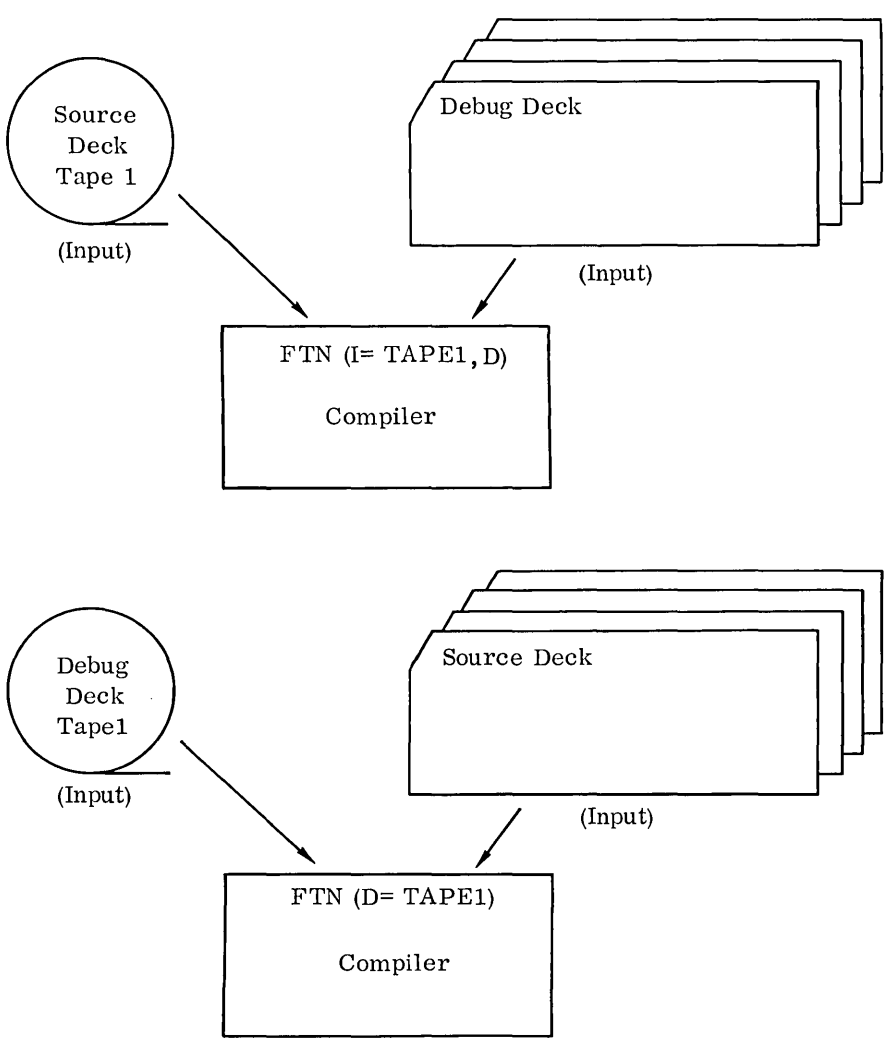
Location	Range
External File	Any or all program units
External Packet	Any or all program units
Internal Packet	Routine containing the packet
Interspersed	Routine containing the specifications

Note: In the following illustrations, it is assumed that a 7/8/9 card terminates each Control Card Record.



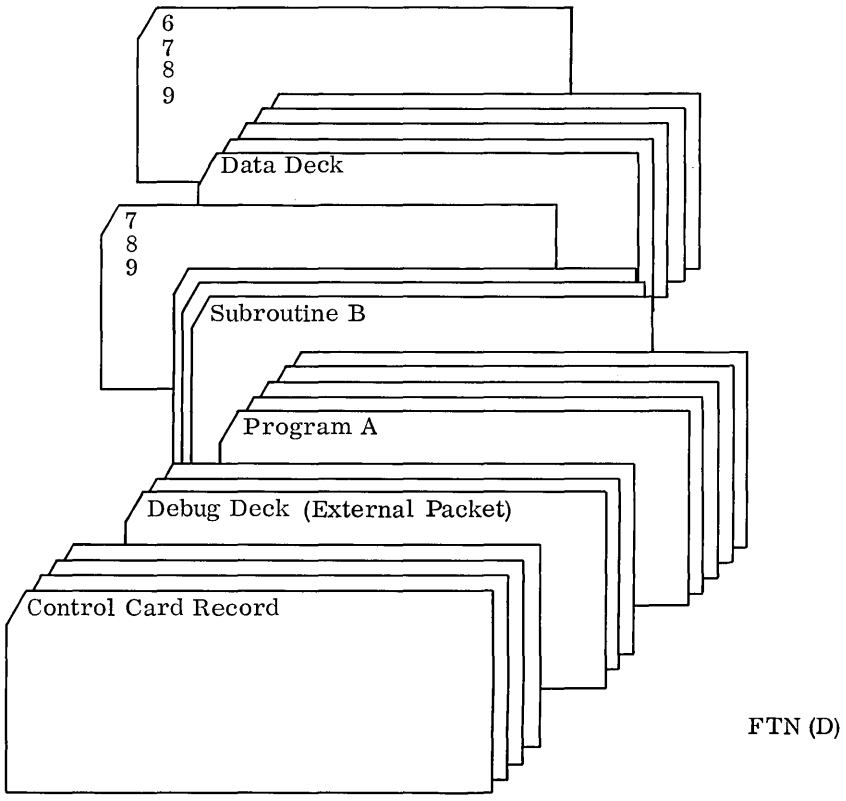
SAMPLE DEBUG AID POSITION: As individual debug cards interspersed in a program unit. The debug cards are inserted into the program where they will be activated. This positioning is especially useful when a new program is to be run for the first time and the accuracy of specific areas, such as array bounds, is in doubt.

Figure 11-1. Sample Debug Aid Position



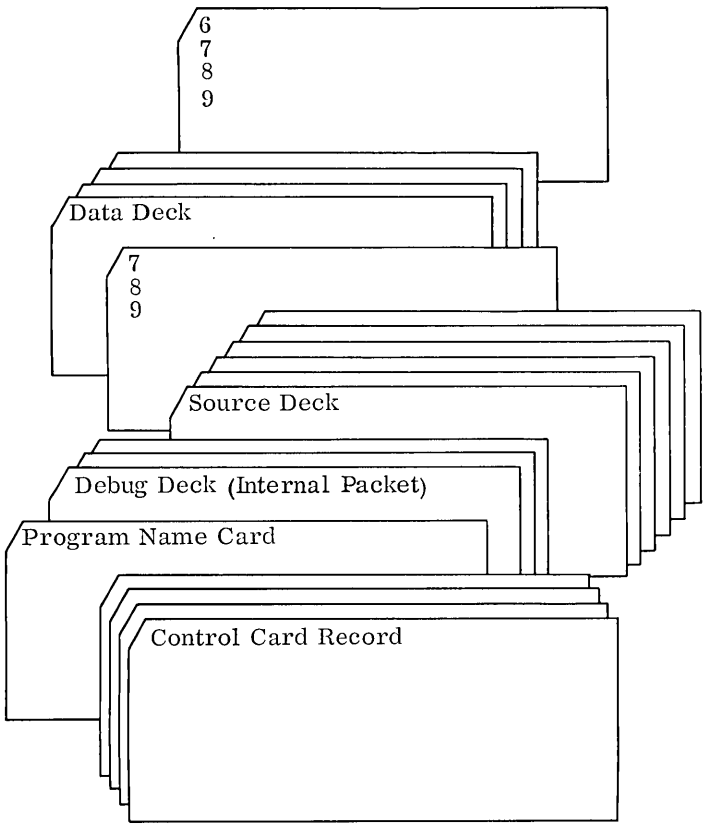
SAMPLE DEBUG AID POSITIONS: Debug deck placed on a separate file (external debug deck) named by the D parameter on the FTN control card, and called in during compilation. With these positions, all program units will be debugged (unless limiting bounds are specified in the deck). This positioning is particularly useful when several jobs can be debugged using the same debugging deck.

Figure 11-2. Sample Debug Aid Positions



SAMPLE DEBUG AID POSITION: As a deck, placed immediately in front of the first source line (when the D file is the same as the source input file). All program units (here, Program A and Subroutine B) will be debugged (unless limiting bounds are specified in the debug deck). This positioning is particularly useful when a program is to be run for the first time, since it ensures that all program units will be debugged.

Figure 11-3. Sample Debug Aid Position



SAMPLE DEBUG AID POSITION: As a deck, placed immediately after the program header card and before any specification statements. All statements in the program unit will be debugged (unless limiting bounds are specified in the debug deck), but no statements in other program units will be debugged. This positioning is especially useful when the job is composed of several program units known to be free of bugs and one unit that is new or is known to have bugs.

Figure 11-4. Sample Debug Aid Position

11.3 DEBUG STATEMENT

A debug deck must begin with a DEBUG statement written in either of the forms:

```
C$   DEBUG

C$   DEBUG(name1,...,namen)
```

The program unit names, to which the debugging deck applies, must be enclosed in parentheses.

In an internal debugging deck, the DEBUG statement must appear immediately after the PROGRAM, SUBROUTINE, or FUNCTION statement heading the routine to which the debugging deck applies. Any names specified in the DEBUG statement, other than the name of the enclosing routine, are ignored.

In a single external debugging deck, whether on the job INPUT file or not, the DEBUG statement may contain a list of the program unit names to which the deck applies. If no name appears, the debugging deck applies to all program units being compiled.

When more than one C$ DEBUG card occurs in an external debugging deck, this card specifies the routines to which the debugging specifications between it and the next C$ DEBUG or non-debugging card apply.

This debug deck specifies arrays checking in all routines, stores checking on the variable CHI routines CHISQ, STATP, and calls checking in routine MAIN.

```
C$   DEBUG

C$   ARRAYS

C$   DEBUG(CHISQ,STATP)

C$   STORES(CHI)

C$   DEBUG(MAIN)

C$   CALLS
```

11.11 AREA STATEMENT

The AREA statement allows a region smaller than a program unit to be debugged. All debugging statements that apply to the program areas designated by the AREA statement must follow that statement. Each succeeding AREA statement cancels the preceding program area designations.

AREA statements may appear only in a debugging deck. If they are interspersed in a FORTRAN source deck, they will be ignored.

The AREA statement can be written in two forms:

C$ AREA(bounds$_1$),...,(bounds$_n$)

for use in a debugging deck with the statement:

C$ DEBUG

or

C$ AREA/name$_1$/(bounds$_1$),.../name$_n$/(bounds$_n$)

for use in a debugging deck with the statement:

C$ DEBUG(name$_1$,...,name$_n$)

C$ DEBUG

The second form of the AREA statement must be used in an external debugging deck.

In the second form of the AREA statement, the /name$_i$/parameter designates the program units to which the bounds following it apply. If a (name$_i$) list appears on the C$ DEBUG card, the /name$_i$/ parameter must be present and name$_i$ must be included in the list. Otherwise, the C$ AREA statement and its associated debugging specifications are ignored. For an external debugging deck the /name$_i$/ field must be present when using either form of the C$ DEBUG statement.

The (bounds) parameter may be written in one of the following forms:

(from field)	indicates line position to be debugged
(from field,to field)	defines a range of line positions which may be in one of the following:
nnnnn	FORTRAN statement label

Lnnnn	Program line number as printed on the source listing (source listing line numbers will change when debugging cards are interspersed in the program.)
id.n	Legal UPDATE line identifier, from the source line, where id = information in columns 73-79; must begin with an alphabetic character and contain no special characters; and n = columns 82-86. (80-81 are blank.)
*	First line in the from field Last line in the to field

A comma must be used to separate the line numbers, and embedded blanks are not permitted.

```
C$   DEBUG(CHISQ)

C$   AREA/CHISQ(210,400)

C$   ARRAYS(SVAL,RMS)

C$   DEBUG(CHISQ,STATP)

C$   AREA/CHISQ/(210,*)/STATP/(L20,L47),

C$   *    (570,L94)

C$   STORES(CHI)

C$   DEBUG(MAIN)

C$   AREA/MAIN/(MAIN.2,MOD1.13)

C$   CALLS
```

11.12 OFF STATEMENT

C$ OFF statements are effective only on interspersed debug directives. In a debugging deck, the C$ OFF statement is ignored.

```
C$   OFF(x1,x2,...,xn)

C$   OFF
```

The C$ OFF statement deactivates subsequent references to debugging options previously activated by interspersed specifications except for C$ NO GO. If a parameter list is specified, only the options in the list are deactivated. Debugging options activated subsequent to the C$ OFF statement and options activated by packet specifications will function normally. The C$ OFF statement is effective at compile time only.

11.13 PRINTING DEBUG OUTPUT

All debug messages produced by the object routines are written to a file named DEBUG. The file always will be printed at job termination time, since it has a print disposition. If the programmer wants to intersperse debug information with his output, he should equate DEBUG to OUTPUT on his program card. A FET and buffer will be supplied automatically at load time if the programmer does not declare the DEBUG file on his program card. For overlay jobs the buffer and FET will be placed in the lowest level of overlay containing debugging. If this overlay level will be overwritten by a subsequent overlay load, the debug buffer will be cleared before it is overwritten.

All object time printing is performed by seven debug routines coded in FORTRAN. These routines are called by code generated when debugging is selected on items such as arrays, calls, stores, etc. The seven routines and their functions are:

ROUTINE	FUNCTION
BUGARR	Checks array subscripts
BUGCLL	Prints messages when subroutines are called. Return
BUGFUN	Prints messages when functions are called. Return
BUGGTA	Prints a message if the target of an assigned GO TO is not in the list.
BUGSTO	Performs stores checking
BUGTRC	Flow trace printing except for true sides of logical IF
BUGTRT	Flow trace printing for true sides of logical IF.

```
 PROGRAM     SAMPLE     DEBUG      TRACE              CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.       PAGE       1

                 PROGRAM SAMPLE(OUTPUT, DEBUG=OUTPUT)
               C$    DEBUG
               C$    AREA(1,100)
               C$    ARRAYS(A1)
05                   DIMENSION A1(5), A2(5)
                   1 PRINT 99
                  99 FORMAT(//* MESSAGES SHOULD FOLLOW FOR REFERENCES TO A1(0) AND
                     **/* A1(6), FOR BOTH LOADS AND STORES.  THERE SHOULD BE NO MESSAGE
                     **/* FOR A2.*)
10                   DO 100 I = 1,4
                     A1(2+I) = A1(4-I) = I
                     A2(2+I) = A2(4-I) = I
                     AGAIN = A1(2+I)
                     AGAIN = A1(4-I)
15                   AGAIN = A2(2+I)
                 100 AGAIN = A2(4-I)
               C$    CALLS(SUB1,SLITE)
                 201 PRINT 299
                 299 FORMAT(//* TWO MESSAGES SHOULD FOLLOW, ONE FOR A CALL OF SUB1 WITH
20                   **/* ARGUMENT 7743, AND ONE FOR A CALL OF SLITE WITH ARGUMENT 1.
                     **/* THERE SHOULD BE NO MESSAGES FOR CALLS OF SUB2 AND SLITET.*)
                     CALL SUB1(7743)
                     CALL SUB2(8242)
                     CALL SLITE(1)
25                 300 CALL SLITET(1,I)
               C$    FUNCS(FUN1,IABS,EXP)
                 301 PRINT 399
                 399 FORMAT(//* MESSAGES SHOULD FOLLOW FOR CALLS OF FUN1 WITH ARGUMENT
                     **/* 7743, IABS WITH ARGUMENT 8242, AND EXP WITH ARGUMENT 3.14159.
30                   **/* THERE SHOULD BE NO MESSAGES FOR CALLS OF FUN2, ABS, OR ALOG.*)
                     IAGAIN = FUN1(7743)
                     IAGAIN = FUN2(7743)
                     IAGAIN = IABS(8242)
                     AGAIN = ABS(8242.)
35                   AGAIN = EXP(3.14159)
                 400 AGAIN = ALOG(3.14159)
               C$    STORES(A1,AGAIN,I,A2.EQ.5.,IAGAIN.LE.10)
                 401 PRINT 499
                 499 FORMAT(//* MESSAGES SHOULD FOLLOW FOR STORES INTO A1(1), A1(2),
40                   **/* A1(3), A1(4), A1(5), I, AGAIN, A2(1), IAGAIN, IAGAIN, IAGAIN,
                     **/* IAGAIN, AND IAGAIN.  THE VALUES STORED IN THE RESPECTIVE
                     **/* VARIABLES SHOULD BE 1., 2., 3., 4., 5., 5, 3.14159, 5.,
                     **/* 10, 9, 8, 7, 6. THERE SHOULD BE NO OTHER STORES MESSAGES.*)
                     DO 402 I = 1,10
45                   A1(I) = I
                     IF(I.EQ.5)GO TO 403
                 402 CONTINUE
                 403 AGAIN = 3.14159
                     DO 500 I = 1,10
50                   A2(1) = 4. + I
                 500 IAGAIN = 16 - I
               C$    AREA(501,600)
               C$    TRACE(3)
                 501 PRINT 599
55               599 FORMAT(//* MESSAGES SHOULD FOLLOW FOR TRANSFERS OF CONTROL
```

```
 PROGRAM     SAMPLE     DEBUG      TRACE              CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE       2

                **/* FROM 502 TO 503, 503 TO 504, 504 TO 505, AND 505 TO 506.
                **/* THERE SHOULD BE NO OTHER CONTROL TRANSFER MESSAGES.*)
                 DO 510 I = 1,2
                 DO 511 J = 1,2
60               DO 512 K = 1,2
                 DO 513 L = 1,2
                 DO 514 M = 1,2
                 GO TO 507
               514 CONTINUE
65             507 GO TO (508,508,508,508),L
               513 CONTINUE
               508 ASSIGN 503 TO L
               502 GO TO L, (503,506)
               512 CONTINUE
70             503 GO TO (504,504), 1
               511 CONTINUE
               504 GO TO 505
               510 CONTINUE
               505 GO TO 506
75             506 CONTINUE
               600 CONTINUE
             C$    OFF
                   A1(1) = 1.
                   GO TO 601
80             601 AGAIN = FUN1(1)
                   I = 1
                   GO TO (602,602),I
               602 CALL SUB1(7743)
                   AGAIN = 3.7
85           C$    GOTOS
               701 PRINT 799
               799 FORMAT(///* WILL NOW ATTEMPT AN ASSIGNED GO TO. SHOULD ISSUE
                  **/* MESSAGE.*)
                   ASSIGN 6327 TO IGO
90                 GO TO IGO, (601,602)
               800 CONTINUE
              6327 PRINT 6328
              6328 FORMAT(///* END OF SAMPLE DEBUG PROGRAM.*)
                   END
```

```
      PROGRAM     SAMPLE    DEBUG     TRACE             CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      3

      SYMBOLIC REFERENCE MAP

ENTRY POINTS
 2026  SAMPLE

VARIABLES     SN  TYPE           RELOCATION
 2723  AGAIN     REAL                          2732  A1        REAL       ARRAY
 2737  A2        REAL       ARRAY              2722  I         INTEGER
 2724  IAGAIN    INTEGER                       2731  IGC       INTEGER
 2725  J         INTEGER                       2726  K         INTEGER
 2727  L         INTEGER                       2730  M         INTEGER

FILE NAMES        MODE
    0  DEBUG                      0  OUTPUT      FMT

EXTERNALS          TYPE   ARGS
       ABS        REAL       1                        ALOG      REAL       1
       EXP        REAL       1                        FUN1      REAL       1
       FUN2       REAL       1                        IABS      INTEGER    1
       SLITE                 1                        SLITET               2
       SUB1                  1                        SUB2                 1

STATEMENT LABELS
    0  1           INACTIVE              2471  99      FMT                     0  100
    0  201         INACTIVE              2510  299     FMT                     0  300         INACTIVE
    0  301         INACTIVE              2534  399     FMT                     0  400         INACTIVE
    0  401         INACTIVE                 0  402                          2205  403
 2561  499     FMT                          0  500                             0  501         INACTIVE
    0  502         INACTIVE              2267  503                          2303  504
 2311  505                               2314  506                          2241  507
 2254  508                                  0  510                             0  511
    0  512                                  0  513                             0  514
 2623  599     FMT                          0  600         INACTIVE         2317  601
 2331  652                                  0  701         INACTIVE         2647  799     FMT
    0  800         INACTIVE              2350  6327                         2660  6328    FMT

STATISTICS
   SYMTAB+DIMTAB     2668     182
  PROGRAM LENGTH     7228     466
   BUFFER LENGTH    20228    1042



      SUBROUTINE  SUB1      DEBUG     TRACE             CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      1

                     SUBROUTINE SUB1(I)
                     END
```

```
      SUBROUTINE  SUB1      DEBUG     TRACE            CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      2

      SYMBOLIC REFERENCE MAP

ENTRY POINTS
    2  SUB1

VARIABLES     SN  TYPE          RELOCATION
    0  I         INTEGER   *UNUSED    F.P.

STATISTICS
   SYMTAB+DIMTAB      38B       24
  PROGRAM LENGTH      7B         7




      SUBROUTINE  SUB2      DEBUG     TRACE            CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      1

                      SUBROUTINE SUB2(I)
                      END




      SUBROUTINE  SUB2      DEBUG     TRACE            CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      2

      SYMBOLIC REFERENCE MAP

ENTRY POINTS
    2  SUB2

VARIABLES     SN  TYPE          RELOCATION
    0  I         INTEGER   *UNUSED    F.P.

STATISTICS
   SYMTAB+DIMTAB      38B       24
  PROGRAM LENGTH      7B         7




      FUNCTION    FUN1      DEBUG     TRACE            CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      1

                      FUNCTION FUN1(I)
                      FUN1 = I
                      END
```

```
      FUNCTION    FUN1      DEBUG     TRACE              CDC 6600 FTN V3.0-P243 OPT=0  01/14/71  16.58.14.      PAGE      2

      SYMBOLIC REFERENCE MAP

ENTRY POINTS
    2  FUN1

VARIABLES     SN  TYPE           RELOCATION
   12  FUN1      REAL                                0  I         INTEGER             F.P.

STATISTICS
   SYMTAB+DIMTAB      32B      26
  PROGRAM LENGTH      13B      11
```

```
      FUNCTION    FUN2      DEBUG     TRACE              CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      1

                      FUNCTION FUN2(I)
                      FUN2 = I
                      END
```

```
      FUNCTION    FUN2      DEBUG     TRACE              CDC 6600 FTN V3.0-P240 OPT=0  01/14/71  16.58.14.      PAGE      2

      SYMBOLIC REFERENCE MAP

ENTRY POINTS
    2  FUN2

VARIABLES     SN  TYPE           RELOCATION
   12  FUN2      REAL                                0  I         INTEGER             F.P.

STATISTICS
   SYMTAB+DIMTAB      32B      26
  PROGRAM LENGTH      13B      11
```

```
CCRE MAP  16.58.37.  NORMAL              CONTROL                                                    013173
          ---TIME---LOAD MODE --L1--L2-----TYPE--------------USER---++---CALL-------------FWA LOAD--LWA LOAD--BLNK COMN--LENGTH--
          FWA LOADER  004771  FWA TABLES  C52456
          -PROGRAM----ADDRESS-                        --LABELED---COMMON--
          SAMPLE      000100
          SUB1        000144
          SUB2        000153
          FUN1        000162
          FUN2        000175
          GETBA       003110
          SIO$        003127
          SYSTEM$     004502
          ACCOER$     005465
          BUGARR$     005510
          BUGCLL$     005570
          BUGCTL$     006177
          BUGFUN$     006134
          BUGGTA$     006553
          BUGSTO$     006671
          BUGTRC$     007715
          DBGFET$     011147
          KODER$      011173
          OUTPTC$     012567
          TRACEX$     012563
          ABS$        012712
          IABS$       012715
          ALNLOGE     012720
          ALOG$       012757
          EXP$        013011
          EXPE        013053
          LEGVAR$     013117
          LOCF$       013124
          SLITE$      013126
          SLITET$     013150
```

DEBUG object time routines

```
         ----UNSATISFIED EXTERNALS-----                              REFERENCES


MESSAGES SHOULD FOLLOW FOR REFERENCES TO A1(0) AND
A1(6), FOR BOTH LOADS AND STORES.  THERE SHOULD BE NO MESSAGE
FOR A2.
/DEBUG/  SAMPLE  AT LINE   11- THE SUBSCRIPT VALUE OF            6 IN ARRAY A1      EXCEEDS DIMENSIONED BOUND OF      5
/DEBUG/          AT LINE   11- THE SUBSCRIPT VALUE OF            0 IN ARRAY A1      EXCEEDS DIMENSIONED BOUND OF      5
/DEBUG/          AT LINE   13- THE SUBSCRIPT VALUE OF            6 IN ARRAY A1      EXCEEDS DIMENSIONED BOUND OF      5
/DEBUG/          AT LINE   14- THE SUBSCRIPT VALUE OF            0 IN ARRAY A1      EXCEEDS DIMENSIONED BOUND OF      5


TWO MESSAGES SHOULD FOLLOW, ONE FOR A CALL OF SUB1 WITH
ARGUMENT 7743, AND ONE FOR A CALL OF SLITE WITH ARGUMENT 1.
THERE SHOULD BE NO MESSAGES FOR CALLS OF SUB2 AND SLITET.
/DEBUG/  SAMPLE  AT LINE   22- ROUTINE SUB1    CALLED  AT LEVEL   0
/DEBUG/          AT LINE   23- ROUTINE SUB1    RETURNS TO LEVEL   0
/DEBUG/          AT LINE   24- ROUTINE SLITE   CALLED  AT LEVEL   0
/DEBUG/          AT LINE   25- ROUTINE SLITE   RETURNS TO LEVEL   0


MESSAGES SHOULD FOLLOW FOR CALLS OF FUN1 WITH ARGUMENT
7743, IABS WITH ARGUMENT 8242, AND EXP WITH ARGUMENT 3.14159.
THERE SHOULD BE NO MESSAGES FOR CALLS OF FUN2, ABS, OR ALOG.
/DEBUG/  SAMPLE  AT LINE   31- REAL    FUNCTION FUN1    CALLED AT LEVEL   0
/DEBUG/          AT LINE   31- REAL    FUNCTION FUN1    RETURNS A VALUE OF   7743.000000     AT LEVEL   0
/DEBUG/          AT LINE   33- INTEGER FUNCTION IABS    CALLED AT LEVEL   0
/DEBUG/          AT LINE   33- INTEGER FUNCTION IABS    RETURNS A VALUE OF            8242 AT LEVEL   0
/DEBUG/          AT LINE   35- REAL    FUNCTION EXP     CALLED AT LEVEL   0
/DEBUG/          AT LINE   35- REAL    FUNCTION EXP     RETURNS A VALUE OF   23.14063123     AT LEVEL   0


MESSAGES SHOULD FOLLOW FOR STORES INTO A1(1), A1(2),
A1(3), A1(4), A1(5), I, AGAIN, A2(1), IAGAIN, IAGAIN, IAGAIN,
IAGAIN, AND IAGAIN.  THE VALUES STORED IN THE RESPECTIVE
VARIABLES SHOULD BE 1., 2., 3., 4., 5., 5, 3.14159, 5.,
10. 9, 8, 7, 6. THERE SHOULD BE NO OTHER STORES MESSAGES.
/DEBUG/  SAMPLE  AT LINE   45- THE NEW VALUE OF THE VARIABLE A1      IS   1.000000000
/DEBUG/          AT LINE   47- THE NEW VALUE OF THE VARIABLE A1      IS   2.000000000
/DEBUG/          AT LINE   49- THE NEW VALUE OF THE VARIABLE A1      IS   3.000000000
/DEBUG/          AT LINE   49- THE NEW VALUE OF THE VARIABLE A1      IS   4.000000000
/DEBUG/          AT LINE   49- THE NEW VALUE OF THE VARIABLE A1      IS   5.000000000
/DEBUG/          AT LINE   48- THE NEW VALUE OF THE VARIABLE AGAIN   IS   3.141590000
/DEBUG/          AT LINE   5 - THE NEW VALUE OF THE VARIABLE A2      IS   5.000000000
/DEBUG/          AT LINE   51- THE NEW VALUE OF THE VARIABLE IAGAIN  IS             10
/DEBUG/          AT LINE   51- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              9
/DEBUG/          AT LINE   51- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              8
/DEBUG/          AT LINE   51- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              7
/DEBUG/          AT LINE   51- THE NEW VALUE OF THE VARIABLE IAGAIN  IS              6


MESSAGES SHOULD FOLLOW FOR TRANSFERS OF CONTROL
FROM 502 TO 503, 503 TO 504, 504 TO 505, AND 505 TO 506.
THERE SHOULD BE NO OTHER CONTROL TRANSFER MESSAGES.
/DEBUG/  SAMPLE  AT LINE   68- CONTROL WILL BE TRANSFERRED TO STATEMENT 503   AT LINE    70
/DEBUG/          AT LINE   7 - CONTROL WILL BE TRANSFERRED TO STATEMENT 504   AT LINE    72
/DEBUG/          AT LINE   72- CONTROL WILL BE TRANSFERRED TO STATEMENT 505   AT LINE    74
/DEBUG/          AT LINE   74- CONTROL WILL BE TRANSFERRED TO STATEMENT 506   AT LINE    75


WILL NOW ATTEMPT AN ASSIGNED GO TO. SHOULD ISSUE
MESSAGE.
/DEBUG/  SAMPLE  AT LINE   9 - ASSIGNED GOTO INDEX CONTAINS THE ADDRESS 002450. NO MATCH FOUND IN STATEMENT LABEL ADDRESS LIST


END OF SAMPLE DEBUG PROGRAM.
```

FORTRAN CONTROL CARD 12

12.1 CONTROL CARD FORMAT

The control card for compilation of a FORTRAN source program consists of the characters FTN and an optional parameter list enclosed in parentheses. If parameters are omitted, FTN is followed by a period. Comments following the right parenthesis or period are transcribed to the dayfile in a normal installation. The first improperly formed parameter terminates the FTN control card scan.

FTN (p_1,p_2,...p_n) comments

or

FTN. comments

When an error is detected in a control card, a dayfile entry is made consisting of an asterisk (below the approximate column in which the compiler encountered the error) and the following message:

*POINTS TO FTN CONTROL CARD ERROR

Example of dayfile:

```
06.52.35.FTN(I=0/L=LIST)
06.52.36.        *
06.52.36. * POINTS TO FTN CONTROL CARD ERROR
```

The job will proceed with the options already processed or terminate and branch to an EXIT(S) card, depending upon an installation option. Default files or files specified in the control card must be in SCOPE 3 format.

12.2 SOURCE INPUT PARAMETER

If the source input parameter is omitted (default condition), the FORTRAN source input file is assumed to be INPUT. If it is on any other file, a parameter of the following form must be provided:

I=lfn (default I=INPUT)

lfn is the logical file name of the file containing the source input. Source input parameters of the forms I=INPUT and I are equivalent to omitting the parameter.

12.3 BINARY (OBJECT) OUTPUT PARAMETER

If the binary output parameter is omitted (default condition), a relocatable binary (object) file is written on a file named LGO. For any other output file, a parameter of the following form must be provided:

B=lfn (default B=LGO)

lfn is the name of the file on which binary output is to be written. Binary output parameters of the form B=LGO or B are equivalent to omitting the parameter.

To suppress production of an object output file, the binary output parameter must be of the form:

B=0

If the letter G appears in the binary output parameter, the object file will be loaded and executed at the end of compilation.

G=lfn BG=lfn GB=lfn G

12.4 LIST PARAMETER

If this parameter is omitted (default condition), a normal listing is provided on OUTPUT; it includes the source program and informative and fatal diagnostics. Other list options may be selected as follows:

y=lfn (default L=OUTPUT,R=1)

y may be one or more of the following:

L	Normal listing
X	Listing of diagnostics which indicate non-ANSI language usage
R	Source keyed cross reference map (implies R=2)
O	Listing of generated object code
N	Suppress listing of informative diagnostics and list only diagnostics fatal to execution

lfn is the file name on which list output is to be written. If lfn is omitted, listing will be on OUTPUT. If L=0 fatal diagnostics with the statements that caused them will be listed; but all other listable output including intermixed COMPASS will be suppressed.

Any combination (with no comma) of the above parameters provides the features indicated. (Note: X and N cannot be used at the same time.)

LRON=lfn specifies all options are to be listed for the file named except non-ANSI diagnostics, and LO selects source and assembly listing on OUTPUT.

CROSS REFERENCE MAP

The FORTRAN Extended cross reference map can be obtained using the R option. This map is described in Appendix C.

12.5 ERROR TRACEBACK AND CALLING SEQUENCE PARAMETER

The T mode of compilation is intended for use with programs in the debugging stages. This parameter is indicated by T†. When it is present, calls to library functions will be made (with the call-by-name sequence), and maximum error checking will be done. Full error traceback will be done if an error is detected.

When T is omitted, the compiler generates calls to library functions with the call-by-value sequence (e.g., cause X1 to contain the parameter, RJ function). Minimum error checking will be done and no traceback will be provided when errors are encountered. A significant saving in memory space and execution time is realized.

12.6 UPDATE PARAMETER (EDITING PARAMETERS)

An E or E=lfn (default E = COMPS) as a parameter requests that the object code output from the compiler produce COMPASS subprogram line images for UPDATE input. This output facilitates hand optimization of the compiler generated object code.

*DECK,name (name = program unit name) is the first card image written on the object code output file, COMPS (assumed when lfn does not appear). An *END card image is written as the last card on the file. COMPASS is not called automatically. The output file lfn or COMPS is rewound and ready for UPDATE input. No binary file is produced.

---

†See Debugging Mode Parameter section in this chapter.
NOTE: The O option is not legal when E is used.

12.7 OPTIMIZATION PARAMETER

The OPT parameter is of the form:

OPT=m

The level of optimization the compiler will perform is determined by the value of m as follows:

m=0	fast compile mode†
m=1	standard compile mode
m=2	fast object code mode

If this parameter is omitted, the installation default option is assumed. Debug mode D option on FTN card implies that OPT=0.

The OPT=2 level of optimization can offer significant execution speed increases for certain classes of loops. Two types of optimization are performed:

- Moving of invariant computations from frequently executed regions to less frequently executed regions.
- Assignment of variables and constants to registers over the body of a loop.

Both DO loops and IF loops can be optimized within these constraints.

- The loop must be the innermost loop (i.e., contain no loops).
- The loop must contain no branching statements (GO TO, IF or RETURN) except a branch back to the start of the loop for IF loops.
- The loop does not contain nonstandard input/output statements such as BUFFER IN/BUFFER OUT, ENCODE/DECODE. In case standard I/O statements occur (or any external calls), only invariant code removal will be attempted.
- Control must flow to the statement following the end of the loop when the loop completes.
- Entry into the loop must be through the sequence of statements preceding the start of the loop.

---

†See Debugging Mode Parameter section in this chapter.

Invariant Computations

In many instances, either for clarity or by accident, calculations which do not change on successive iterations are made within a loop. When these computations are made outside the loop, the speed of the loop is improved without changing the results.

Example 1:

```
      DO 100 I = 1,2000
100 A(I) = 3*I + J/K+5
```

A more efficient loop would be:

```
      ITERM = J/K+5
      DO 100 I = 1,2000
100 A(I) = 3*I + ITERM
```

For clarity, the programmer may not wish to write the code in this form. Using the OPT=2 level, the more efficient loop structure would be produced. A message will be issued stating:

```
n WORDS OF INVARIANT RLIST REMOVED FROM
  THE LOOP STARTING AT LINE x
```

RLIST is the intermediate language of the compiler. The message serves two functions. It notifies the programmer that his loop has been modified, and it informs him of the magnitude of the change.

Example 2:

```
      I = 1
200 J = K+L+4
      A(I) = M+I
      I = I+1
      IF(I. LE. 100) GO TO 200
```

Use of OPT=2 will produce code as if Example 2 had been written as shown below:

```
      I = 1
      J = K+L+4
200   A(I) = M+I
      I = I+1
      IF(I.LE.100) GO TO 200
```

Example 3:

```
      DO 300 I=1,2000
      A(I) = SQRT(FLOAT(I))
      A(I) = A(I) + 3.5*R
300   CONTINUE
```

The computation of 3.5*R will be removed from the loop in spite of the external call. In general, this process will occur unless R is a parameter to the external routine or in COMMON. The use of a variable will not be recognized as invariant if it is a member of an equivalence class for which some member of the class is referenced inside the loop using nonstandard subscripts. For standard subscripts, optimization will occur, although the assumption is made that all subscripting is within the bounds of dimensional declarations.

Register Assignment

For many loops, it is possible to keep commonly used variables and constants in the registers. Eliminating loads and stores from the body of the loop has two advantages:

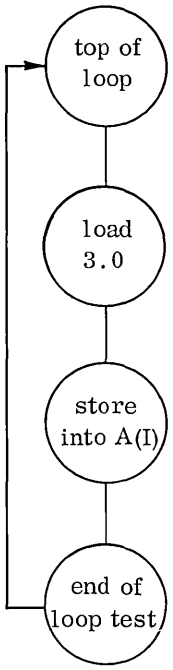
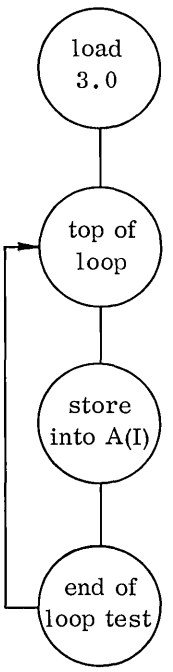
- Reducing the number of loads and stores increases the execution speed.
- The loop is shortened and may fit in the instruction stack of the 6600.

Presently up to four X registers may be assigned over a loop. The actual number assigned depends on the number of candidates available for selection and the complexity of the operations performed within the loop. When registers are assigned, an informative message is printed:

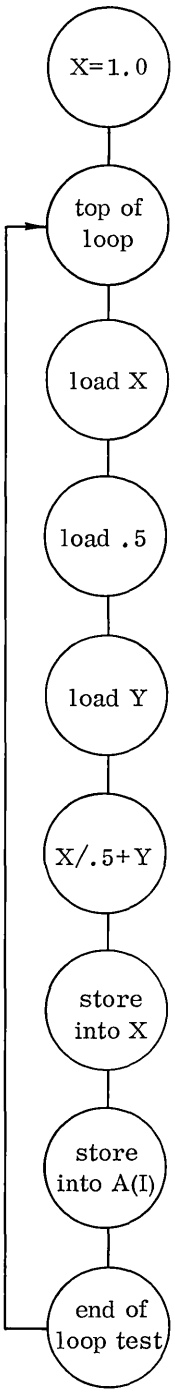
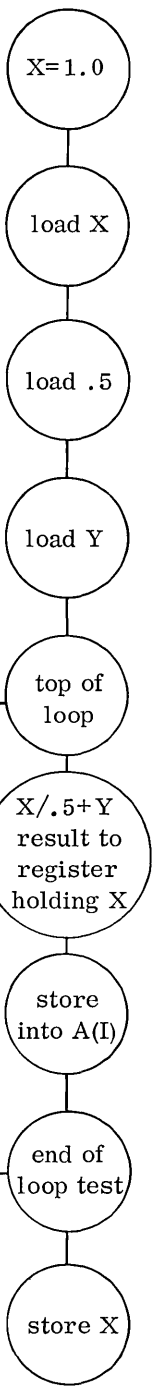
```
n REGISTERS ASSIGNED OVER THE LOOP BEGINNING AT LINE x
```

Register assignment will not be performed for loops containing external references.

Example 1:

Loop	Without assignment	With assignment
DO 100 I=1,2000 A(I) = 3.0 100 CONTINUE		

Example 2:

Loop	Without assignment	With assignment
X = 1.0 DO 200 I=1,100 X = X/.5+y A(I) = X 200 CONTINUE		

12.8 ROUNDED ARITHMETIC PARAMETER

The compiler will produce rounded arithmetic instructions for any combination of arithmetic operators (+ - * /) if the parameter is specified in the form:

ROUND=operators (default=OFF)

If this parameter is omitted (default condition), rounded arithmetic processing does not take place.

12.9 DEBUGGING MODE PARAMETER

When this parameter is selected, the OPT=0 compilation and T error traceback modes are assumed. If the debugging mode parameter is omitted (default condition), this mode of compilation does not take place.

D or D=lfn (default = INPUT)

lfn specifies the file name of the debugging aid selection package.

12.10 EXIT PARAMETER

When this parameter is specified, the run will terminate and branch to an EXIT(S) control card if fatal errors occur during compilation. The form is:

A (default off)

12.11 SYSTEM TEXT FILE PARAMETER

The S parameter specifies the systems text file to be used for intermixed COMPASS programs.

S = 0 or S = lfn (default lfn = SYSTEXT)

If S=0 when COMPASS is called to assemble any intermixed COMPASS programs, it will not read in a system text file. If this parameter is omitted (default condition), S=SYSTEXT is assumed.

12.12 SYSTEM EDITING AND I/O REFERENCE PARAMETER

This parameter is of the following form:

SYSEDIT=FILES

SYSEDIT=IDENT

SYSEDIT (default, both FILES and IDENT)

The FILES specification assures that all INPUT/OUTPUT references will be accomplished indirectly through GETBA. In addition, the file names are not entry points in the main program, and subprograms do not produce external references to the file names. When IDENT is specified, a $ is appended to the program name on both the IDENT and ENTRY cards if the program name is the same as that of any FORTRAN object library program.

12.13 ASSEMBLER PARAMETER

The COMPASS assembler, rather than the FTN built-in assembler, can be used to assemble the code generated by FTN. The COMPASS assembler is specified with the following parameter:

C (default off)

12.14 CONTROL CARD EXAMPLES

The control card FTN. is equivalent to:

FTN (I=INPUT,L=OUTPUT,B=LGO,S=SYSTEXT,OPT=1,R=1)

The control card:

FTN (A,LRN,G,S=0)

will select the following options:

A — Abort, branch to EXIT(S) card when errors occur in compilation

LRN — List on the file OUTPUT, which will include a source-keyed cross reference map, and suppress the informative diagnostics.

G — Placed on file LGO, the relocatable binary file. If compilation is successful, it will be loaded and executed.

S=0 — When COMPASS is called to assemble intermixed COMPASS subprograms, it will not read in a system text file.

12.15 SMALL BUFFERS

When this option is selected, the compiler is forced to use 513-word buffers for compiler intermediate files. Programs with a large number of declarations may be compiled with a smaller field length if this parameter is specified. Since less space is used in the buffers, compile time may increase. The form of the parameter is:

V (default = off)

12.16 REFERENCE MAP LEVEL

The kind of reference map produced is determined by the R option on the control card:

R = 0 No map

1 Short map (symbols, addresses, properties)

2 Long map (short map, references by line number and a DO-loop map)

3 Long map and printout of common block members and equivalence classes

blank Implies R = 1

The default option is R = 1 unless the L option equals 0; then R = 0.

APPENDIX SECTION

CDC 6500

DISPLAY CODE	CHARACTER	DISPLAY CODE	CHARACTER
00	blank	45	+
01	A	46	-
02	B	47	*
03	C	50	/
04	D	51	(
05	E	52	)
06	F	53	$
07	G	54	=
10	H	55	blank
11	I	56	, (comma)
12	J	57	. (period)
13	K	60	≡
14	L	61	[
15	M	62	]
16	N	63	: (colon)
17	O	64	≠
20	P	65	→
21	Q	66	v
22	R	67	∧
23	S	70	↑
24	T	71	↓
25	U	72	<
26	V	73	>
27	W	74	≤
30	X	75	≥
31	Y	76	¬
32	Z	77	; (semicolon)
33	0		
34	1		
35	2		
36	3		
37	4		
40	5		
41	6		
42	7		
43	8		
44	9		

SOURCE PROGRAM CHARACTERS A

FORTRAN CHARACTERS

Alphabetic Characters	Console Display Code	Hollerith Card Punch
A	01	12-1
B	02	12-2
C	03	12-3
D	04	12-4
E	05	12-5
F	06	12-6
G	07	12-7
H	10	12-8
I	11	12-9
J	12	11-1
K	13	11-2
L	14	11-3
M	15	11-4
N	16	11-5
O	17	11-6
P	20	11-7
Q	21	11-8
R	22	11-9
S	23	0-2
T	24	0-3
U	25	0-4
V	26	0-5
W	27	0-6
X	30	0-7
Y	31	0-8
Z	32	0-9
Numeric Characters		
0	33	0
1	34	1
2	35	2
3	36	3
4	37	4
5	40	5
6	41	6
7	42	7
8	43	8
9	44	9

Special Characters	Console Display Code	Hollerith Card Punch
+	45	12
-	46	11
/	50	0-1
=	54	8-3
,	56	0-8-3
(	51	0-8-4
$	53	11-8-3
*	47	11-8-4
.	57	12-8-3
)	52	12-8-4
blank	55	space

ADDITIONAL CHARACTERS

Character	Console Display Code	Hollerith Card Punch
≡	60	0-8-6
[	61	8-7
]	62	0-8-2
:	63	8-2
≠	64	8-4
→	65	0-8-5
∨	66	11-0†
∧	67	0-8-7
↑	70	11-8-5
↓	71	11-8-6
<	72	12-0††
>	73	11-8-7
≤	74	8-5
≥	75	12-8-5
¬	76	12-8-6
;	77	12-8-7
end-of-line	0000	

†11-0 and 11-8-2 are equivalent
††12-0 and 12-8-2 are equivalent

Diagnostic messages are produced by the FORTRAN processor to inform the user of errors in the program. Messages are produced during compilation and execution; compilation errors are discussed in this appendix, a detailed discussion of the execution errors is given in Appendix G.

Errors detected during compilation are noted on the source listing, immediately following the END card. Figure B-1 illustrates a listing and the format used by the processor in noting compilation errors.

```
      100 WRITE (6,8)
        8 FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
          119X,1H1/19X,1H3)
      101 I=5
 5      8 A=I
      102 A=SQRT(A)
      103 J=A
      104 DO 1 K=3,J,2
      105 L=I/K
10    106 IF(L*K-I)1,2,4
        1 GO TO 108
      107 WRITE (6,9)
        5 FORMAT (I20)
        2 I=I+2
15    108 IF(1000-I)7,4,3
        4 WRITE (6,7)
        9 FORMAT (14H PROGRAM ERROR)
        7 WRITE (6,6)
        6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
20    109 STOP
          END
```

CARD NO.	SEVERITY		DIAGNOSTIC
01	I	START.	ASSUMED PROGRAM NAME WHEN NO HEADER STATEMENT APPEARS
05	FE	8	THIS STATEMENT NUMBER HAS BEEN USED BEFORE
11	FE		A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT
21	FE	7	THIS REFERENCED FORMAT NUMBER DOES NOT APPEAR ON A FORMAT STATEMENT
25	FE	3	THIS REFERENCED STATEMENT NUMBER DOES NOT APPEAR ON AN EXECUTABLE STATEMENT

Figure B-1. Sample Source Listing

The source of the errors is identified by the card number. This number corresponds to the card number assigned by the processor indicated by the numbers on the extreme left side of the example. The severity of the error is indicated by the code accompanying the message: I means informative and has no effect on compilation or execution, FE indicates catastrophic to execution, FC means the error is catastrophic to compilation, and USASI indicates that the particular usage does not conform to USASI standards. USASI diagnostics are not listed unless requested by an X parameter on the FTN control card.

The diagnostics that follow are printed in full capitals, for readability, they are shown here in upper and lower case.

FC Errors

Compiler error at ER21A or ER21B of DO processor.

Tables overlap, more memory required.

Auxiliary table overflow.

Arithmetic statement function has caused a table overflow while being processed. More memory required.

Symbol table overflow. Reduce number of variables.

CONLIST too big, too many constants.

Tables overlap during equivalence processing. More memory required.

Insufficient memory.

The statement beginning at line is too complex for this compiler. Please simplify it.

Not enough room in working storage to hold all OVERLAY control card information.

FE Errors

Loops are nested more than 50 deep.

The terminal label of a DO must be an integer constant between 0 and 100,000.

The terminal statement of this DO precedes it.

The control variable of a DO or DO implied loop must be a simple integer variable.

The syntax of DO parameters must be $I=M_1,M_2,M_3$ or $I=M_1,M_2$.

A constant DO parameter must be between 0 and 131K.

A DO parameter must be an integer constant or variable.

This statement number has been used before.

A previous statement in this nest references this statement number illegally.

This statement references a previous statement number in this nest illegally.

A DO loop may not terminate on this type of statement.

A DO loop which terminates here includes one or more unterminated DO loops.

Entry statements may not occur within the range of a DO statement.

This DO loop is unterminated at program end.

This loop is entered from outside its range but has no exit.

This referenced statement number does not appear on an executable statement.

This referenced FORMAT number does not appear on a FORMAT statement.

Program card delimiter missing.

Filename is greater than 6 characters.

Filename previously defined.

Unit number or parity indicator must be an integer constant or variable.

Equated filename not previously defined.

Unrecognized statement.

Illegal label field in next statement.

Statement too long.

Symbolic name has too many characters.

Unmatched parenthesis.

Unlabeled FORMAT statement.

Duplicate statement label.

RETURNS list error.

Doubly defined formal parameter.

No legal list terminator.

Illegal separator between variables.

Variable has more than three subscripts.

Variable with illegal subscript.

Variable dimension is not a formal parameter.

Variable in common has either an adjustable subscript or is a formal parameter.

Header card not first statement.

Common block name not enclosed in slashes.

COMMON variable is formal parameter or previously declared in COMMON.

Illegal block name.

Illegal separator in EXTERNAL statement.

A reference to this arithmetic statement function was not followed by an open parenthesis.

Insufficient memory, possibly a recursively defined arithmetic statement function.

A reference to an improperly specified arithmetic statement function has been encountered.

A reference to this arithmetic statement function has unbalanced parenthesis within the parameter list.

Unmatched parameter count in a reference to this statement function.

A constant cannot be converted. Check constant for proper construct.

RETURN statement appears in main program.

Non-standard RETURN statement may not appear in a function subprogram.

Parameter on non-standard RETURN statement is not a RETURNS formal parameter.

Illegal sequence in I/O list.

FORMAT reference must be an integer constant or an array reference.

Entry point names must be unique — this one has been previously used in this subprogram.

Improper form of ENTRY statement, only allowable form is [entry name] .

Referenced label is more than five characters.

ENTRY statement is not allowed to appear within a program, only in a subroutine or function.

There is an entry in this namelist statement other than a slash, a comma, or a variable.

NAMELIST name is either not a variable or a variable that has been previously defined.

NAMELIST group name is not surrounded by slashes.

— this entry appeared in a position where a variable should have appeared.

Name of a variable that is not allowed to be used in conjunction with NAMELIST.

— this variable has variable dimensions, this is not allowed in conjunction with NAMELIST.

Statement number is not allowed on an ENTRY statement.

There is insufficient room to process this statement, more memory required.

Formal parameters or ECS variables cannot appear in EQUIVALENCE statements.

Subscripts not integer constants. Equivalencing abandoned.

Only one symbolic name in EQUIVALENCE group.

Syntax error in EQUIVALENCE statement.

Subscript value is out of range of the array as determined by the dimensions.

COMMON-EQUIVALENCE error.

Number of subscripts is incompatible with the number of dimensions during EQUIVALENCEing.

Common block origin extended, extension not allowed.

Symbol was involved in contradictory equivalencing. Equivalencing abandoned.

Either the expected list of transfer labels is non-existent, empty, or not enclosed in parentheses.

This is not a recognizable form of the GO TO statement.

There is a non numeric entry in this list of transfer labels.

Number of characters in an ENCODE/DECODE statement must be an integer constant or variable.

More than 50 files on program card or 63 parameters on subroutine or function card.

Declarative statement out of sequence.

Error table overflow.

This ASSIGN statement has improper format, only allowable is [ASSIGN no. to variable].

Illegal identifier in variable list of DATA statement.

Variable appearing in DATA statement may not be in blank COMMON.

Variable appearing in DATA statement may not be a formal parameter.

Variable appearing in DATA statement may not be a function name.

Illegally typed variable in DATA statement must be only integer, real, double, complex, or logical.

Illegal format of DATA statement.

All items in data list of DATA statement must be constant.

Repeat factor of DATA items and DO limits must be integer.

Constant subscript of variable must be integer.

No terminating right parenthesis after subscript or DO variables.

DO control variable not used as subscript in DATA statement.

No equal sign after DO variable in DATA statement.

Implied DO loop may have only 3 limits.

Variable appeared as subscript but its DO limits were never defined.

Non dimensioned identifier appears with subscripts in DATA statement.

Unmatched parentheses in DATA statement.

Zero statement labels are illegal.

Illegal character after DATA item, must be comma, slash, or left paren.

Only a comma or end of statement may follow terminating slash or right parenthesis in a DATA statement.

Slash, equal sign, or left parenthesis must follow variable list.

Illegal use of the equal sign.

Variable followed by left parenthesis.

No matching right parenthesis.

No matching left parenthesis.

The operator indicated (-, +, *, /, or **) must be followed by a constant, name, or left parenthesis.

More than 63 arguments in argument list.

A constant may not be followed by an equal sign.

Expression translator table OPSTAK overflowed, simplify the expression.

Logical operand used with non-logical operators.

No matching right parenthesis in subscript.

Local entry point referred to as external function.

Statement function reference may not use a function name as an argument.

Argument not followed by comma or right parenthesis.

A function reference requires an argument list.

Illegal CALL format.

Expression translator table FRSTB overflowed, simplify the expression.

Illegal input/output address.

Right parenthesis followed by a name, constant, or left parenthesis.

More than one relational operator in a relational expression.

A comma, left paren, =, .OR., or .AND. must be followed by a name, constant, left paren, -, .NOT., or +.

An array reference has too many subscripts.

No matching right parenthesis in argument list.

Illegal form involving the use of a comma.

Logical and non-logical operands may not be mixed.

Division by constant zero.

A complex base may only be raised by an integer.

Use of this program or subroutine name in an expression.

Subroutine name referred to by CALL is used elsewhere as a non-subroutine name.

Illegal call format.

Illegal returns parameter.

Illegal labels in IF statement.

Logical expression in 3-branch IF statement.

The statement in a logical IF may be any executable statement other than a DO or another logical IF.

The expression in a logical IF is not type logical.

Left side of replacement statement is illegal.

A reference to this ASF has a parameter missing.

All elements in an ECS common block must be type ECS.

A previously mentioned adjustable subscript is not type integer.

All ECS variables must appear in an ECS common block.

The type of this identifier is not legal for any expression.

A constant operand of a real operation is out of range or indefinite.

Referenced label is greater than five characters.

This combination of operand types is not allowed in this version.

Implied DO in I/O statement is unterminated, check paren count and syntax.

_____ was last character seen before trouble. Remainder of statement was skipped.

Double or complex operand in subscript expression not allowed.

Double or complex argument not legal for this intrinsic function.

No terminating right parenthesis in OVERLAY, SEGMENT, SEGZERO or SECTION card.

I Errors

This statement redefines a current loop control variable or parameter.

More storage required by DO statement processor for optimization.

The variable upper limit and the control variable of this DO are the same producing a non-terminating loop.

The constant lower limit is greater than the constant upper limit of a DO

No END card. End line assumed.

START. assumed program name when no header statement appears.

Undefined variable, i.e., this variable is never initialized.

Previously dimensioned variable, first dimensions will be retained.

Previously typed variable, first encountered type is retained.

Dummy parameter in an arithmetic statement function definition occured twice

Arithmetic statement function has more dummy parameters than are allowed (20).

There is an entry following the right parentheses of this assigned GO TO list.

In this unconditional GO TO there is an entry following the transfer statement label.

More data items appear in data list than array can contain, excess items are discarded.

More memory would have resulted in better optimization.

Array name operand not subscripted; first element will be used.

The number of arguments in the argument list of a non-basic external function is inconsistent.

The number of arguments in a subroutine argument list is inconsistent.

Number of digits in constant exceed possible significance. High order digits retained when possible.

USASI Usage Diagnostics

Dummy parameter in an arithmetic statement function definition occurred twice.

Arithmetic statement function has an improperly formed parameter list or no = following the list.

The non-standard RETURN statement is not USASI FORTRAN.

This statement is non-USASI.

Non-USASI form of DATA statement.

More than one equal sign.

Array name referenced with fewer subscripts than the dimensionality of the array.

FORMAT Statement Validation Diagnostic Messages

The word preceding the diagnostic has the following form:

xx cd nnnn

where

xx is a card column number

nnnn is a card number

Example:

67 cd 5 refers to column 67 of card 5

Informative Diagnostics:

Separator missing, separator assumed here.

X field preceded by a blank, 1X assumed.

X field preceded by a zero, no spacing occurs.

Preceding field width is zero.

Preceding field width should be 7 or more.

Floating point descriptor expects decimal point specified. Output will include no fractional parts.

Floating point specification expects decimal digits to be specified. Zero decimal digits assumed.

Repeat count for preceding field descriptor is zero.

Field width is outside inner limits. Check use of this format to assure device can handle this record size.

Preceding scale factor is outside limits of representation within the machine.

Superfluous scale factor encountered preceding current scale factor.

Record size outside inner limits. Check use of this format to assure device can handle this record size.

Field width of preceding floating point descriptor should be 7 or more than decimal digits specified.

Numeric field following tab setting designator is equal to zero, column one is assumed.

Numeric field omitted in preceding scale factor. Zero scale assumed.

Non-blank characters follow zero-level right parenthesis. These characters will be ignored.

Tab setting may exceed record size depending on use.

USASI Usage Diagnostics:

Plus sign is an illegal character.

Preceding field descriptor is non-USASI.

Floating point descriptor expected following scale factor designator.

Tab setting designator is non-USASI.

Hollerith string delineated by symbols is non-USASI.

Fatal to Execution Diagnostics:

Preceding character illegal at this point in character string. Error scan for this format stops here.

Illegal character follows preceding floating point descriptor. Error scan for this format stops here.

Illegal character follows preceding A, I, L, O, or R descriptor. Error scan for this format stops here.

Illegal character follows tab setting designator. Error scan for this format stops here.

Illegal character follows preceding sign character. Error scanning for this format stops here.

Preceding character illegal, scale factor expected. Error scanning for this format stops here.

Preceding Hollerith count is equal to zero. Error scanning for this format stops here.

Format statement ends before last Hollerith count is complete. Error scan for this format stops at H.

Format statement ends before end of Hollerith string. Error scanning stops here.

Preceding Hollerith indicator is not preceded by a count. Error scanning stops here with format incomplete.

Zero level right parenthesis missing. Scanning stops.

Preceding field width outside outer limits for record size. Scanning continues.

Preceding record outside outer limits for record size. Scanning continues.

Tab setting is outside outer limits for record length. Scanning continues.

FORTRAN Extended Assembler Diagnostics:

Storage overflow, no object program will be produced.

Increase field length by xxxxx.

CROSS REFERENCE MAP C

The cross reference map is a dictionary of all programmer created symbols appearing in a program unit, with the properties of each symbol and references to each symbol listed by source line number. The symbol names are grouped by class and are listed alphabetically within the groups. The reference map begins on a separate page following the source listing of the program and the error dictionary.

The kind of reference map produced is determined by the R option on the control card:

R = 0	No map
1	Short map (symbols, addresses, properties)
2	Long map (short map, references by line number and a DO-loop map)
3	Long map and printout of common block members and equivalence classes
blank	Implies R = 1

The default option is R = 1 unless the L option equals 0; then R = 0.

Errors in the source program will cause certain parts of the map to be suppressed, incomplete, or inaccurate. Fatal execution and fatal compilation errors will cause the DO-loop map to be suppressed, and assigned addresses will be different; symbol references may not be accumulated for statements containing syntax errors.

For the long map, it may be necessary to increase field length by 1000(octal).

The number of references that can be accumulated and sorted at reference map time is: field length - 20000(octal)-4 x (number of symbols). For a source program containing one thousand symbols, approximately eight thousand references can be accumulated with a field length of 50000 octal.

In the following pages, examples from the actual cross-reference map produced by the debugging program reproduced in chapter 11 are interspersed with the general format discussions. The complete cross reference map and the generating program may be found at the conclusion of chapter 11.

General Format

Formats for each symbol class are different, but printouts for all the classes contain the following information:

Program or common relative address of the symbol (in octal with leading zeros suppressed)

The symbol as it appears in the FORTRAN source listing

Properties associated with the symbol

List of references to the symbol

All line numbers in the reference list refer to the line of the statement in which the reference occurs. Multiple references in a statement are printed as n*l where n is the number of references on line l.

All numbers to the right of the name are decimal integers unless they are suffixed with a B which indicates octal.

Names of symbols generated by the compiler (such as system library routines called for input/ ouput) do not appear in the reference map.

CLASSES OF SYMBOLS

Each class of symbol is preceded by a subtitle line that specifies the class and the properties listed.

Entry Points

Entry point symbols include subprogram names and names appearing in ENTRY statements. The subtitle line is printed:

ENTRY POINTS DEFINITION REFERENCES

RA Name Def Ref

RA Program relative address

Name Entry point name as it appears in the FORTRAN source

Def Line number of subprogram statement or line on which ENTRY statement occurs

Ref Line number (none for a main program). In a subprogram, RETURN statements are references to the entry point. (For a function subprogram, references to the function value appear in the variable map.)

```
ENTRY POINTS
 2026  SAMPLE
```

Example:

```
      SUBROUTINE SUBR
      COMMON A, B, C
      IF(A .EQ. 0.0) GO TO 10
      RETURN
   10 B = C**2
      A = B+C
      RETURN
      END
```

The subtitle line and properties are:

```
ENTRY POINTS     DEFINITION     REFERENCES
   2 SUBR              1             4   7
```

Variables

Variable symbols include local and common variables and arrays, formal parameters, RETURNS names, and for a function subprogram, the function name when used as a variable. The subtitle line is:

```
VARIABLES     SN     TYPE          RELOCATION
 RA   Name     *      Type     Properties     Blockname     (Refs, Defined, IO Refs)
```

RA — Program or common relative address; 0 for formal parameters

Name — Variable name as it appears on the source listing

SN — Stray name flag. Names which are variable names and appear only once in a subprogram are indicated by *. Variables in this category are stray, since they may be keypunch errors, misspellings, etc. A legal usage that would cause a stray name is a DO loop where the control variable is not referenced. (Present only for R = 2 or R = 3)

Type — The arithmetic mode of a variable (logical, integer, real, double precision, complex, or ECS). RETURNS is printed for RETURNS formal parameters. Types are offset to aid in debugging.

Properties — The following keywords may be printed out in this column:

*UNDEF — The symbol has not been defined. A symbol is defined if any of the following conditions hold:

It appears in a COMMON or DATA statement

It is a non-base member of an equivalence class

It appears on the left side of an assignment statement, at the outermost parenthesis level

It is the control variable in a DO loop

It appears as a stand alone actual parameter in a subroutine or function call

It appears in an input I/O list

Variables used before definition are not detected

ARRAY Symbol is dimensioned.

*UNUSED Name is an unused formal parameter. If nothing is printed and the name is not a RETURNS parameter, it is a simple variable.

Blockname

blank Local symbol (address is program unit relative)

F·P· Formal parameter

/ / Symbol is in blank common, RA relative to blank common

name Name of common block where symbol appears; RA is common relative

Refs, Defined, IO Refs

References and definitions are the rightmost items in a variable map.

REFS References are collected for variable symbol names appearing in declarative statements or used in assignment statements.

DEFINED Definitions are listed for names appearing in DATA statements, the control variable of a DO loop, names defined in an ASSIGN or assignment statement, and names defined by READ or ENCODE/DECODE statements. The subprogram header line defines formal parameters.

IO REFS Input/output references are collected for symbols used as variable file names in an I/O statement.

In a function subprogram, references to the function name are listed in the variable map.

References are collected after statement functions are expanded and are not collected for the arguments before expansion.

Example: If ASF (J) = (J + 1)/(J - 1) is a statement function and K = ASF (I) is on line 5: two references will be listed for I on line 5.

```
VARIABLES      SN  TYPE            RELOCATION
 2723  AGAIN      REAL                                2732  A1      REAL      ARRAY
 2737  A2         REAL        ARRAY                   2722  I       INTEGER
 2724  IAGAIN     INTEGER                             2731  IGO     INTEGER
 2725  J          INTEGER                             2726  K       INTEGER
 2727  L          INTEGER                             2730  M       INTEGER
```

File Names

File names include those used as logical file names (unit number) in the input/output statements or names declared as files on the program card in a main program. The subtitle line is printed:

```
FILE NAMES          MODE
FET    Name     Mode   Reference
```

FET Program relative address of the file environment table associated with the file. The file's buffer starts at FET + 17D (listed in a main program and blank in subroutine).

Name Filename

Mode One of the following will be printed:

blank	If the mode cannot be determined from operations on the file
UNFMT	Unformatted I/O, no conversion
FMT	Formatted I/O
BUF	Buffer I/O
MIXED	Some combination of the above

References are divided into the categories:

READS	Input operations
WRITES	Output operations
MOTION	Positioning operations; rewind/backspace and ENDFILE

```
FILE NAMES          MODE
    0  DEBUG                          0  OUTPUT       FMT
```

External References

External references include names of subroutines or functions external to a subprogram. Names of system routines not explicitly called in the source program such as those used for input/output and exponentiation will be suppressed. The subtitle line is:

```
EXTERNALS     TYPE      ARGS      REFERENCES
  Name        Type    Args Prop      Refs
```

Name Symbol name as it appears in the FORTRAN source

Type

blank	Subroutine
NO TYPE	Conversion will follow the same rule as for octal or Hollerith data
Other	Arithmetic mode

Args Number of arguments used to reference symbol

Prop

Blank	Programmer defined function or subroutine
F·P·	Formal parameter
LIBRARY	Call by value library function

Refs Lines on which symbol was referenced

In T or D mode, no LIBRARY functions appear since they are all called by name and most intrinsic functions are compiled as external references.

```
EXTERNALS          TYPE   ARGS
        ABS        REAL      1          ALOG       REAL       1
        EXP        REAL      1          FUN1       REAL       1
        FUN2       REAL      1          IABS       INTEGER    1
        SLITE                1          SLITET                2
        SUB1                 1          SUB2                  1
```

Inline Functions

Inline functions include names of intrinsic and statement functions appearing in the subprogram. The subtitle line is:

```
INLINE FUNCTIONS     TYPE     ARGS     DEF     LINE     REFERENCES
      Name           Mode     Args     Ftype   Def      Refs
```

Name	Symbol name as it appears in the listing	
Mode	Arithmetic mode, NO TYPE means no conversion in mixed mode expressions	
Args	Number of arguments with which the function is referenced	
Ftype	INTRIN	Intrinsic function
	SF	Statement function
Def	Blank for intrinsic functions; the definition line for statement functions	
Refs	Lines on which function is referenced	

Namelist Group Names

This listing includes names declared as namelist group names in NAMELIST statements. The subtitle line is:

```
NAMELISTS     DEF LINE          REFERENCES
   Name       Line of definition  References to group name
```

Statement and Format Labels

The label map includes all statement labels appearing in the program. Labels may be referenced in input/output, GO TO, ASSIGN, IF or CALL statements with RETURNS lists. The subtitle line is:

```
STATEMENT LABELS          DEF LINE     REFERENCE
RA  label  Type  Activity      Def          Refs
```

RA label	Program relative address. Inactive labels are printed with a zero address.	
Type	Blank	Executable statement number
	FMT	Format number
	UNDEF	Label is undefined

Activity	Blank	Label is active or referenced
	INACTIVE	Label is inactive (statement number)
	NO REFS	Label is defined as a format number and not referenced
Def	Line number in which label appeared in columns 1-5 of a source statement	
Refs	Lines in which label was referenced	

Active labels are those for which the compiler has not deleted all references by optimization.

The following example contains the only reference to the label 5 in a program. The label is inactive because the compiler deletes jumps to the next statement.

```
      IF(X)10,5,10
 5    X = 1
10    continue
```

Labels referenced only in DO statements as loop terminators are not assigned addresses.

Inactive labels and those used as loop terminators cannot be assigned any meaningful address by the compiler.

```
STATEMENT LABELS
    0  1          INACTIVE    2471  99      FMT                    0  100
    0  201        INACTIVE    2510  299     FMT                    0  300         INACTIVE
    0  301        INACTIVE    2534  399     FMT                    0  400         INACTIVE
    0  401        INACTIVE       0  402                         2205  403
 2561  499    FMT                0  500                            0  501         INACTIVE
    0  502        INACTIVE    2267  503                         2303  504
 2311  505                    2314  506                         2241  507
 2254  508                       0  510                            0  511
    0  512                       0  513                            0  514
 2623  599    FMT                0  600          INACTIVE       2317  601
 2331  602                       0  701          INACTIVE       2647  799     FMT
    0  800        INACTIVE    2350  6327                        2660  6328    FMT
```

DO-Loop Maps

This map is a printout of all DO loops appearing in the source program and their properties. The map may be generated by the R = 2 or R = 3 option. Loops are listed in order of their appearance in the program. The subtitle line is:

LOOPS	LABEL	INDEX	FROM-TO	LENGTH	PROPERTIES
fwa	term mf	index	lf-lt	len	prop

fwa	First word address of loop body	
term	Label associated with end of loop, or blank for I/O loop	
mf	*	Loop index is kept in memory to generate code for loop counting mechanism

	blank	Other
index	Variable used to control loop	
lf-lt	Numbers of first and last lines of loop	
len	Number of instruction words generated for body of loop	
prop	If loop can be optimized, one of these messages is printed:	
	OPT	Loop has no properties which inhibit optimization
	INSTACK	Loop is seven words or less, compiler assembled in 6600 mode
	If loop is not optimized by the compiler, the reasons are listed:	
	EXT REFS	Loop contains references to an external subroutine or function, or it is an input/output loop
	ENTRIES	Loop is entered from outside its range
	EXITS	Loop contains references to labels outside its range
	NOT INNER	Loop is not innermost loop in a nest

Loops that fit in the 6600 instruction stack have a maximum length of seven words, and usually run two to three times as fast as a comparable loop that does not fit in the stack.

Common Blocks

Common block symbols include common block names and names declared in common statements to be variables and arrays in common. The subtitle line is printed:

```
COMMON BLOCKS     LENGTH     MEMBERS - BIAS NAME (LENGTH)
    bname           blen          bias     mname    (size)
```

bname	Block name
blen	Total block length

When the common block members are to be printed (R = 3), the following details appear for each member declared in a COMMON statement.

bias	Common relative address (distance from block origin)
mname	Member name
size	Number of words allocated for member

If an equivalence class is linked to common, all members of the class become members of the common block. These members are listed in the equivalence class printout.

Equivalence Classes

This class of symbol is collected only when R = 3. All members of an equivalence class explicitly mentioned in EQUIVALENCE statements are listed. Any symbols added through linkage to common are not included. The subtitle is:

```
EQUIV    CLASSES    MEMBERS - BIAS    NAME    (LENGTH)
 pbase     base         clen    bias   mname    (size)
```

pbase	*ERROR*	Class is in error (more than one member in common or block origin extended by equivalence)
	base member	Class in common
	blank	Other
base	If the class is local, base is the name of the base member of the class, the one with the smallest address. (If the class is in common, the name of the symbol in common which linked the equivalence class to the common block is printed.) When an equivalence class is in common, the base member of the equivalence class is the first member of the common block.	
clen	Class length or span	
bias	Distance from the class base to the member	
mname	Member name	
size	Number of words allocated for the member	

Members of a class are printed in the order of increasing bias. If the class is in error, the numbers associated with the class length and bias are meaningless.

Program Statistics

At the end of the reference map, the statistics are printed in octal and decimal. The subtitle line is:

STATISTICS

program length	Program length including code, storage for local variables, arrays, constants, temporaries, etc., but excluding buffers and common blocks.
buffer length	Total space occupied by I/O buffers and FETs
common length	Total common length, excluding blank common
blank common	Length of blank common

```
STATISTICS
  PROGRAM LENGTH     722B      466
   BUFFER LENGTH   2022B     1042
```

Error Messages

The following error messages are printed if sufficient storage is not available:

CANT SORT THE SYMBOL TABLE INCREASE FL BY NNNB

or

REFERENCES AFTER LINE NNN LOST INCREASE FL BY NNNB

DEBUGGING (USING THE REFERENCE MAP)

New Program:

The reference map can be used to find names that have been punched incorrectly as well as other items that will not show up as compilation errors. The basic technique consists of using the compiler as a verifier and correcting the FE errors until the program compiles.
Using the listing, the R=3 reference map, and the original flowcharts, the following information should be checked by the programmer:

Names incorrectly punched

Stray name flag in the variable map

Functions that should be arrays

Functions that should be inline instead of external

Variables or functions with incorrect mode

Unreferenced format statements

Unused formal parameters

Ordering of members in common blocks

Equivalence classes

Existing Program:

The reference map can be used to understand the structure of an existing program. Questions concerning the loop structure, external references, common blocks, arrays, equivalence classes, input/output operations, and so forth, can be answered by checking the reference map.

Intrinsic Function & No. of Arguments	Definition	Example	Symbolic Name	Type of Argument	Type of Function
Absolute value (1)	|a|	Y=ABS(X)	ABS	Real	Real
		J=IABS(I)	IABS	Integer	Integer
		DOUBLE A, B B=DABS(A)	DABS	Double	Double
Truncation (1)	trunc (a) = [a] if a≥0 -[-a] if a<0 where the function represented by [a] is defined to be the integer i that satisfies i≤ a< i+1	Y=AINT(X)	AINT	Real	Real
		I=INT(X)	INT	Real	Integer
		DOUBLE Z J=IDINT(Z)	IDINT	Double	Integer
Modulo	MOD or AMOD (a_1, a_2) is defined to be $a_1 - \text{trunc}(a_1/a_2)*a_2$	B=AMOD(A1, A2)	AMOD	Real	Real
		J=MOD(I1, I2)	MOD	Integer	Integer
		DM=DMOD(D1, D2)	DMOD	Double	Double
Choosing largest value (≥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 (≥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

Intrinsic Function & No. of Arguments	Definition	Example	Symbolic Name	Type of Argument	Type of Function
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
			DSIGN	Double	Double
Positive difference (2)	a_1 - Min(a_1,a_2)	Z=DIM(X, Y)	DIM	Real	Real
		J=IDIM(I1,I2)	IDIM	Integer	Integer
Truncate to obtain most significant part of double precision argument (1)		DOUBLE Y X=SNGL(Y)	SNGL	Double	Real
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 single precision argument in double precision form (1)		DOUBLE Y Y=DBLE(X)	DBLE	Real	Double
Express two real arguments in complex form (2)	$a_1 + a_2 \sqrt{-1}$	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
Shift (2)	Shift a_1 by a_2 bit positions: left circular if a_2 is positive; right with sign extension if a_2 is negative	B=SHIFT(A,I)	SHIFT	a_1: Single word a_2: Integer	Octal
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)	$\neg a$	B=COMPL(A)	COMPL	Single word	Octal
▌Masking		MASK(I)	MASK	Integer	Octal

External Function & No. of Arguments	Definition	Example	Symbolic Name	Type of Argument	Type of 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)	$\sqrt{AIMAG^2(a)+REAL^2(a)}$	CM=CABS(CX)	CABS	Complex	Real
Arccosine (1)	arccos (a)	X=ACOS(Y)	ACOS	Real	Real

External Functions & No. of Arguments	Definition	Example	Symbolic Name	Type of Argument	Type of Function
Arcsine (1)	arcsin (a)	X=ASIN(Y)	ASIN	Real	Real
Trigonometric tangent (1)	tan (a)	Y=TAN(X)	TAN	Real	Real
Random number generator (1)	ranf (a) returns values uniformly distributed over the range [0,1)	X=RANF(DUM)	RANF	Dummy	Real
Address of argument a (1)	loc (a)	P=LOCF(X)	LOCF	Symbolic	Integer
I/O status on buffer unit (1)	= -1 unit ready; no error = 0 EOF on last operation = +1 parity error	IO=UNIT(6)	UNIT	Integer	Real
I/O status on non-buffer unit (1)	= 0 no EOF in previous read	IFL=EOF(4)	EOF	Integer	Real
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

† These routines may be used as functions or subroutines. The value is always returned via the argument and via the normal function return.

Subroutine & No. of Arguments	Definition	Example	Symbolic Name	Type of Argument
Set Sense Light (1)	$1 \leq i \leq 6$ turn sense light is on. i = 0 turn off all sense lights.	CALL SLITE(I)	SLITE	Integer
Test Sense Light (2)	If sense light i is on j = 1. If off j = 2 Always turn sense light i off	CALL SLITET(I, J)	SLITET	Integer
Test Sense Switch (2)	If sense switch i is down j = 1. If sense switch i is up j = 2.	CALL SSWTCH(I, J)	SSWTCH	Integer
Terminate (0)	Terminate program execution and return control to the monitor	CALL EXIT	EXIT	
Console Comment (1)	Place a message of up to 80 characters on dayfile†	CALL REMARK (2HHI)	REMARK	Hollerith
Console Value (2)	Display up to a 10 character message and value in the dayfile†	CALL DISPLA (2HX=,20.2)	DISPLA	a_1=Hollerith a_2=real or integer
Obtain current generative value of RANF between 0 and 1 (1)	ranget (a)	CALL RANGET(X)	RANGET	Symbolic
Initialize generative value of RANF (1)	ranset (a), the generic value is set to the nearest odd number ≥ a	CALL RANSET(X)	RANSET	Real
Dump memory (3-60)	dump(a, b, f)	CALL DUMP(A, B, 1)	DUMP	Logical
	dump A to B according to f	CALL PDUMP(X,Y,0)	PDUMP	Integer Real Double Complex
Input checking (2)	ERRSET (a,b), set maximum number of errors, b, allowed in input data before fatal termination. Error count kept in a.	CALL ERRSET(A, B)	ERRSET	Symbolic Integer

†Characters with a display code value above 57_8 are not allowed. The message must be terminated with binary zeros, even if an entire word is necessary. (Use of a Hollerith constant of any form will provide such a termination automatically.)

INTERMIXED COMPASS SUBPROGRAMS E

Subprograms written in COMPASS may be intermixed with FORTRAN coded subprograms in the source deck. COMPASS subprograms must begin with a card containing the word IDENT in columns 11-15, and terminate with card containing the word END in columns 11-13. Columns 1-10 of the IDENT and END cards must be blank; column 14 of the END card must be blank.

Calling Sequence

When the FORTRAN compiler encounters a reference to an external subprogram, subroutine, or function the following calling sequence is generated:

```
SA1     Argument list (if parameters appear)
RJ      Subprogram name
```

where the argument list consists of consecutive words of the form:

```
VFD 60/argument_i
```

followed by a zero word.

Control Return

The COMPASS subprogram must restore the initial contents of A0 in A0 upon returning control to the calling subprogram. When the COMPASS subprogram is entered via a function reference, the result of that function must be in X6 or X6 and X7 with the least significant or imaginary part of the double precision or complex result appearing in X7.

Statements	Classification	Page
Entry Points		
PROGRAM s	N†	9-1
PROGRAM s ($f_1,f_2,\ldots,f_n$)	N	9-1
SUBROUTINE s	N	9-2
SUBROUTINE s ($a_1,a_2,\ldots,a_n$)	N	9-2
SUBROUTINE s, RETURNS ($b_1,b_2,\ldots,b_m$)	N	9-2
SUBROUTINE s ($a_1,a_2,\ldots,a_n$),RETURNS ($b_1,b_2,\ldots,b_m$)	N	9-2
FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
REAL FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
DOUBLE FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
COMPLEX FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
INTEGER FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
LOGICAL FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
DOUBLE PRECISION FUNCTION f ($a_1,a_2,\ldots,a_n$)	N	9-8
ENTRY s	N	9-5
Specification Program Declaration		
BLOCK DATA	N	9-10
BLOCK DATA d	N	9-10
Inter-subroutine		
EXTERNAL $v_1,v_2,\ldots,v_n$	NS	8-7
Inter-subroutine Transfer Statements		
CALL s	E	4-12
CALL s ($a_1,a_2,\ldots,a_n$)	E	4-12
CALL s, RETURNS ($b_1,b_2,\ldots,b_m$)	E	4-12

† N=Non-executable, S=Specification, E=Executable.

Statements (Cont'd)	Classification	Page
CALL s (a_1,a_2,...,a_n), RETURNS (b_1,b_2,...,b_m)	E	4-12
RETURN	E	4-14
RETURN a	E	4-14

Data Declaration and Storage Allocation

Type Declaration

REAL v_1,v_2,...,v_n	NS	8-7
DOUBLE v_1,v_2,...,v_n	NS	8-7
COMPLEX v_1,v_2,...,v_n	NS	8-7
INTEGER v_1,v_2,...,v_n	NS	8-7
LOGICAL v_1,v_2,...,v_n	NS	8-7
DOUBLE PRECISION v_1,v_2,...,v_n	NS	8-7
ECS v_1,v_2,...,v_n	NS	8-7
TYPE REAL v_1,v_2,...,v_n	NS	8-7
TYPE DOUBLE v_1,v_2,...,v_n	NS	8-7
TYPE COMPLEX v_1,v_2,...,v_n	NS	8-7
TYPE INTEGER v_1,v_2,...,v_n	NS	8-7
TYPE LOGICAL v_1,v_2,...,v_n	NS	8-7
TYPE DOUBLE PRECISION v_1,v_2,...,v_n	NS	8-7
TYPE ECS v_1,v_2,...,v_n	NS	8-7

Storage Allocation

DIMENSION $v_1(i_1)$, $v_2(i_2)$,...,$v_n(i_n)$	NS	8-1
COMMON /x_1/a_1/.../x_n/a_n	NS	8-3
EQUIVALENCE (k_1), (k_2),...,(k_n)	NS	8-6
DATA k_1/d_1/, k_2/d_2/,...,k_n/d_n/	N	8-8
DATA (r_1=d_1), (r_2=d_2),...,(r_n=d_n)	N	8-10

Statement Function

f (a_1,a_2,...,a_n) = e	E	9-7

Symbol Manipulation and Control	Classification	Page
Replacement Statements		
v=e { Arithmetic	E	3-1
v=e { Logical	E	3-3
v=e { Masking	E	3-3
Intra-program Transfers		
GO TO k	E	4-1
GO TO i, (k_1,k_2,...,k_n)	E	4-1
GO TO (k_1,k_2,...,k_n), e	E	4-2
IF (e) k_1,k_2,k_3	E	4-3
IF (e) k_1,k_2	E	4-4
IF (e) s	E	4-4
Loop Control		
DO n i = m_1,m_2,m_3	E	4-5
Miscellaneous Program Controls		
ASSIGN k TO i	E	4-1
CONTINUE	E	4-12
PAUSE	E	4-15
PAUSE n	E	4-15
STOP	E	4-14
STOP n	E	4-14
Input/Output		
I/O Format		
FORMAT ($q_1t_1z_1t_2z_2...t_nz_nq_2$)	N	6-1

I/O Control Statements	Classification	Page
READ f,k	E	5-3
READ (u) k	E	5-4
READ (u)	E	5-4
READ (u,f) k	E	5-2
READ (u,f)	E	5-2
WRITE (u) k	E	5-4
WRITE (u,f)	E	5-3
WRITE (u,f) k	E	5-3
PRINT f,k	E	5-4
PUNCH f,k	E	5-4
BUFFER IN (u,p) (A,B)	E	7-2,I-2
BUFFER OUT (u,p) (A,B)	E	7-2,I-2
Internal Manipulation		
ENCODE (n,f,A) k	E	I-6,7-3
DECODE (n,f,A) k	E	I-6,7-3
Tape Handling		
ENDFILE u	E	5-10
REWIND u	E	5-9,I-5
BACKSPACE u	E	5-9,I-5
Miscellaneous		
NAMELIST $/y_1/a_1/y_2/a_2/.../y_n/a_n$	N	5-6
Program Termination		
END	N	4-15

The SYSTEM routine handles error tracing, diagnostic printing, termination of output buffers, and transfer to specified non-standard 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 zero 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 Frequence 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 every n^{th} time until the print limit is reached.

Fatal (F)/ Non-Fatal (NF)

If the error is non-fatal and a non-standard 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 non-standard 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 non-standard recovery address is specified, see Non-Standard Recovery.

Non-Standard Recovery

SYSTEM supplies the non-standard recovery routine with the following information:

- A1 Address of parameter list passed to the routine which detected the error
- X1 Address of the first parameter
- A0 Address of parameter list of the routine that called the routine which detected the error
- B1 Address of a secondary parameter list, which contains, in successive words:
  - Error number passed in SYSTEM
  - Address of diagnostic word available to SYSTEM
  - Address within auxiliary table if A/NA bit is set, otherwise zero
  - Instruction consisting of RJ to SYSTEM in upper 30 bits and trace back information in lower 30 bits for the routine that called SYSTEM
- A2 Address of error table entry in SYSTEM
- X2 Contents of error table entry

Information in the secondary parameter list is not available to FORTRAN-coded routines.

Non-Fatal Error

The routine which detected the error and SYSTEM are delinked from the calling chain and the non-standard recovery routine is entered. When this routine exits in the normal routine, control returns to the routine which called the routine which detected the error.

Thus, any faulty arguments may be corrected, and the recovery routine is allowed to call the routine which detected the error, providing corrected arguments. By not correcting the faulty arguments in the recovery routine, a three routine loop can develop between the routine which detects the error, SYSTEM, and the recovery routine. No checking is done for this case.

Fatal Error

SYSTEM calls the non-standard recovery routine in the normal fashion, with the registers set as indicated above. If the non-standard 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 non-standard 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/non-fatal (fatal = 1, non-fatal = 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 Non-standard 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

   (dddd is FORTRAN source line count)

ERROR SUMMARY
   ERROR      TIMES
   xxxxx      yyyy
              .
              .
              .
     (all numbers are decimal)
```

zzzzzzz and cccccc are routine names, yyyyyy and wwwwww are relocatable addresses

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 non-standard error recovery address, 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.

Execution Diagnostics

Routine	Message	Error No.
ACGOER$	ERROR IN COMPUTED GO TO STATEMENT: INDEX VALUE INVALID	1
ACOS$	ABS(R).GT.1.0 INFINITE ARGUMENT INDEF ARGUMENT	2
ALOG$	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	3
ALOG10$	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	4
ASIN$	ABS(R).GT.1.0 INFINITE ARGUMENT INDEF ARGUMENT	5
ATAN$	INFINITE ARGUMENT INDEF ARGUMENT	6
ATAN2$	X=Y=0.0 INFINITE OR INDEF ARGUMENT	7
CABS$	FLOATING OVERFLOW INFINITE OR INDEF ARGUMENT	8
ZTOI$	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE OR INDEF ARGUMENT	9
CCOS$	INFINITE OR INDEF ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	10
CEXP$	INFINITE OR INDEF ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	11
CLOG$	ZERO ARGUMENT INFINITE OR INDEF ARGUMENT	12
COS$	ARG TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEF ARGUMENT	13

Routine	Message	Error No.
CSIN$	INFINITE OR INDEF ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	14
CSQRT$	INFINITE OR INDEF ARGUMENT	15
DABS$	INFINITE ARGUMENT INDEF ARGUMENT	16
DATAN$	INFINITE ARGUMENT INDEF ARGUMENT	17
DATAN2$	X=Y=0.0 INFINITE OR INDEF ARGUMENT	18
DTOD$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DBLE POWER INFINITE OR INDEF ARGUMENT	19
DTOI$	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE OR INDEF ARGUMENT	20
DTOZ$	FLOATING OVERFLOW IN D**REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(D) TOO LARGE INFINITE OR INDEF ARGUMENT	21
DTOX$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DBLE POWER INFINITE OR INDEF ARGUMENT	21
DCOS$	ARG TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEF ARGUMENT	22
DEXP$	ARGUMENT TOO LARGE, FLOATING OVERFLOW INFINITE ARGUMENT INDEF ARGUMENT	23

Routine	Message	Error No.
DLOG$	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	24
DLOG10$	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	25
DMOD$	DP INTEGER EXCEEDS 96 BITS 2ND ARGUMENT ZERO INFINITE OR INDEF ARGUMENT	26
DSIGN$	INFINITE ARGUMENT INDEF ARGUMENT	27
DSIN$	ARG TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEF ARGUMENT	28
DSQRT$	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	29
EXP$	ARGUMENT TOO LARGE, FLOATING OVERFLOW INFINITE ARGUMENT INDEF ARGUMENT	30
ITOJ$	INTEGER OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER	31
IDINT$	INTEGER OVERFLOW INFINITE OR INDEF ARGUMENT	32
XTOD$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DBLE POWER INFINITE OR INDEF ARGUMENT	33
XTOI$	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE OR INDEF ARGUMENT	34

Routine	Message	Error No.
XTOY$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE OR INDEF ARGUMENT	35
SIN$	ARG TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEF ARGUMENT	36
SLITE$	ILLEGAL SENSE LITE NUMBER	37
SLITET$	ILLEGAL SENSE LITE NUMBER	38
SQRT$	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEF ARGUMENT	39
SSWTCH$	ILLEGAL SENSE SWITCH NUMBER	40
TAN$	ARG TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEF ARGUMENT	41
TANH$	INFINITE ARGUMENT INDEF ARGUMENT	42
ITOD$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DBLE POWER INFINITE OR INDEF ARGUMENT	44
ITOX$	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE OR INDEF ARGUMENT	45
ITOZ$	FLOATING OVERFLOW IN I**REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(I) TOO LARGE INFINITE OR INDEF ARGUMENT	46

Routine	Message	Error No.
XTOZ$	FLOATING OVERFLOW IN X**REAL(Z)	47
	ZERO TO THE ZERO OR NEGATIVE POWER	
	NEGATIVE TO THE COMPLEX POWER	
	IMAG(Z)*LOG(X) TOO LARGE	
	INFINITE OR INDEF ARGUMENT	
FTNERR$	COMPILATION ERROR ENCOUNTERED DURING PROGRAM EXECUTION	48
INPUTN$	TOO FEW CONSTANTS FOR UNSUBSCRIPTED ARRAY	49
OVERLA$	FATAL ERROR IN LOADER	50
SEGMEN$	FATAL ERROR IN LOADER	51
	NON-FATAL ERROR IN LOADER	52
BACKSP$	UNASSIGNED MEDIUM, FILE NAME: xxxxxxx	53
BUFFEI$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	54
	END-OF-FILE ENCOUNTERED, FILENAME: xxxxxxx	55
	WRITE FOLLOWED BY READ ON FILE: xxxxxxx	56
	BUFFER DESIGNATION BAD--FWA.GT.LWA	57
BUFFEO$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	58
	BUFFER SPECIFICATION BAD--FWA.GT.LWA	59
ENDFIL$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	60
IFENDF$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	61
INPUTB$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	62
	END-OF-FILE ENCOUNTERED, FILENAME xxxxxxx	63
INPUTO$ OUTPTN$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	64
INPUTC$	END-OF-FILE ENCOUNTERED, FILENAME: xxxxxxx	65
INPUTN$	PRECISION LOST IN FLOATING INTEGER CONSTANT	66
	NAMELIST DATA TERMINATED BY EOF, NOT $	
	NAMELIST NAME NOT FOUND	
	NO I/O MEDIUM ASSIGNED	
	WRONG TYPE CONSTANT	
	INCORRECT SUBSCRIPT	
	TOO MANY CONSTANTS	
	(,$, OR = EXPECTED, MISSING	
	VARIABLE NAME NOT FOUND	
	BAD NUMERIC CONSTANT	
	MISSING CONSTANT AFTER *	
	UNCLEARED EOF ON A READ	
	READ PARITY ERROR	

Routine	Message	Error No.
INPUTS$	*DECODE*CHAR/REC.GT./50*	66
IOCHECK$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	67
KODER$	*ILLEGAL FUNCTIONAL LETTER	68
	*IMPROPER PARENTHESIS NESTING	69
	*EXCEEDED RECORD SIZE	70
	*SPECIFIED FIELD WIDTH ZERO	71
	*FIELD WTH .LE. DECIMAL WTH	72
	*HOLLERITH FORMAT WITH LIST	73
KRAKER$	*ILLEGAL FUNCTIONAL LETTER	74
	*IMPROPER PARENTHESIS NESTING	75
	*SPECIFIED FIELD WIDTH ZERO	76
	*EXCEEDED RECORD SIZE	77
	*ILLEGAL DATA IN FIELD *↓*	78
	*DATA OVERFLOW *>*	79
	*HOLLERITH FORMAT WITH LIST	80
LENGTH$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	81
FTNBIN$ OUTPTB$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	82
OUTPTC$ CONNEC$	UNASSIGNED MEDIUM, FILENAME: xxxx	83
OUTPTN$	OUTPUT FILE LINE LIMIT EXCEEDED	84
OUTPTS$	ENCODE*CHAR/REC .GT. 150*	85
REWINM$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	86
KODER$	*LIST/FMT CONFLICT, SNGL/DBLE	87
INPUTB$	WRITE FOLLOWED BY READ ON FILE: xxxxxxx	88
	LIST EXCEEDS DATA, FILENAME: xxxxxxx	89
	PARITY ERROR READING (BINARY) FILE: xxxxxxx	90
INPUTC$	WRITE FOLLOWED BY READ ON FILE: xxxxxxx	91
	PARITY ERROR READING (CODED) FILE: xxxxxxx	92
OUTPTB$	PARITY ERROR ON LAST READ ON FILE: xxxxxxx	93
OUTPTC$	PARITY ERROR ON LAST READ ON FILE: xxxxxxx	94

Routine	Message	Error No.
IOCHEC$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	95
	*STATUS OF BUFFER I/O MUST BE CHECKED BY THE UNIT FUNCTION * FILENAME: xxxxxxx	96
INITMS$ READMS$ WRITMS$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	97
INITMS$	FILE DOES NOT RESIDE ON A RANDOM ACCESS DEVICE	98
READMS$	FILE WAS NOT OPENED BY A CALL TO SUBROUTINE OPENMS	99
WRITEMS$		
READMS$	RECORD NAME REFERRED TO IN CALL IS NOT IN THE FILE INDEX	100
INITMS$ WRITMS$	INDEX BUFFER IS OF INSUFFICIENT LENGTH	101
LABEL$	UNASSIGNED MEDIUM, FILENAME: xxxxxxx	102
READMS$	*READ PARITY ERROR*	102
READMS$	SPECIFIED INDEX IN THIS MASS STORAGE CALL .GT. MASTER INDEX OR IS ZERO	110
WRITEC$	ECS UNIT HAS LOST POWER OR IS IN MAINTENANCE MODE	112
READEC$	ECS READ PARITY ERROR	113

DECK STRUCTURE H

Program Unit Structure

FORTRAN Extended program unit source decks are divided into five sections as follows; they must conform to the order shown.

Section	Content
A	Program unit identification (PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA card)
B	Specification statements (DIMENSION, TYPE, etc.)
C	Statement function definition
D	Executable statements (X=Y, GOTO14, etc)
E	END statement

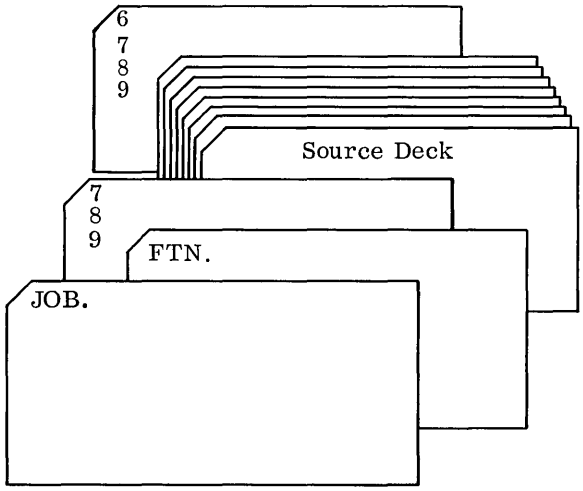
Section A should appear in every program. Sections B, C and D may include FORMAT statements and comment lines. Sections C and D may include NAMELIST and DATA statements. If Section A is a BLOCKDATA statement, Sections C and D may not be included in the program unit. Section E should appear if multiple subprograms are used, since if no recognizable header card is present on the following subprogram, a fatal error occurs.

Source Decks

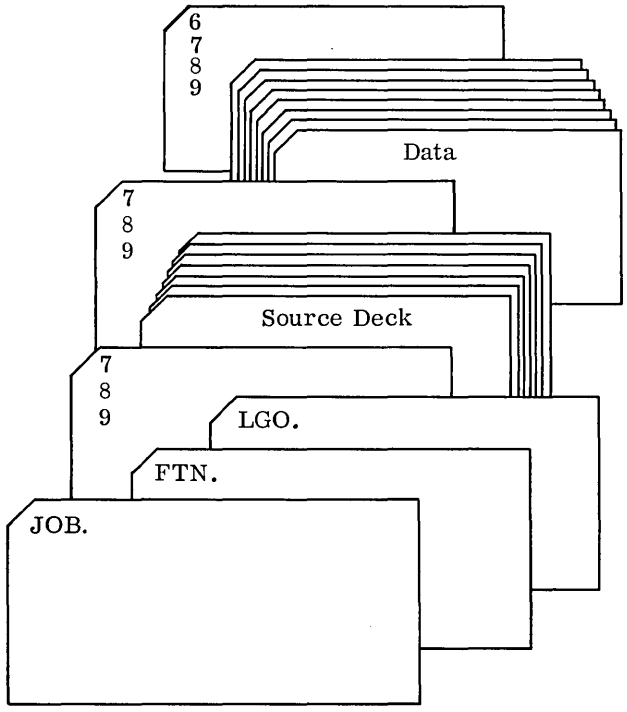
Source decks are comprised of complete FORTRAN program unit source decks and/or COMPASS source decks. Each COMPASS source deck must begin with an IDENT card (columns 11-15) and terminate with an END card (columns 11-13); in both cases columns 1-10 must be blank. FORTRAN and COMPASS program unit source decks may be in any order.

SAMPLE DECK STRUCTURE

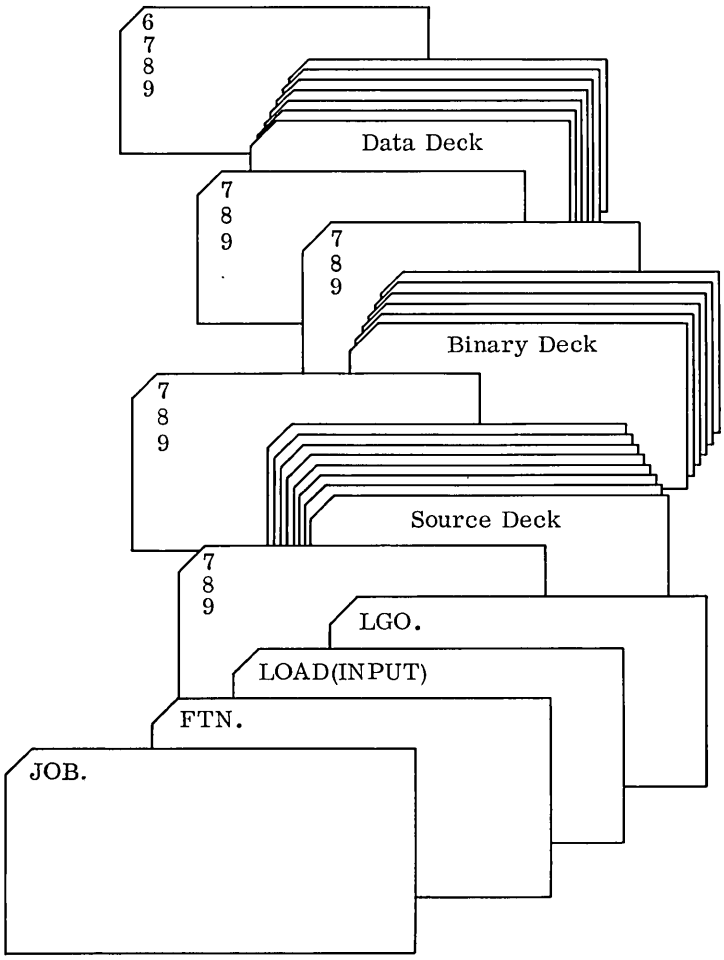
1. Compilation only



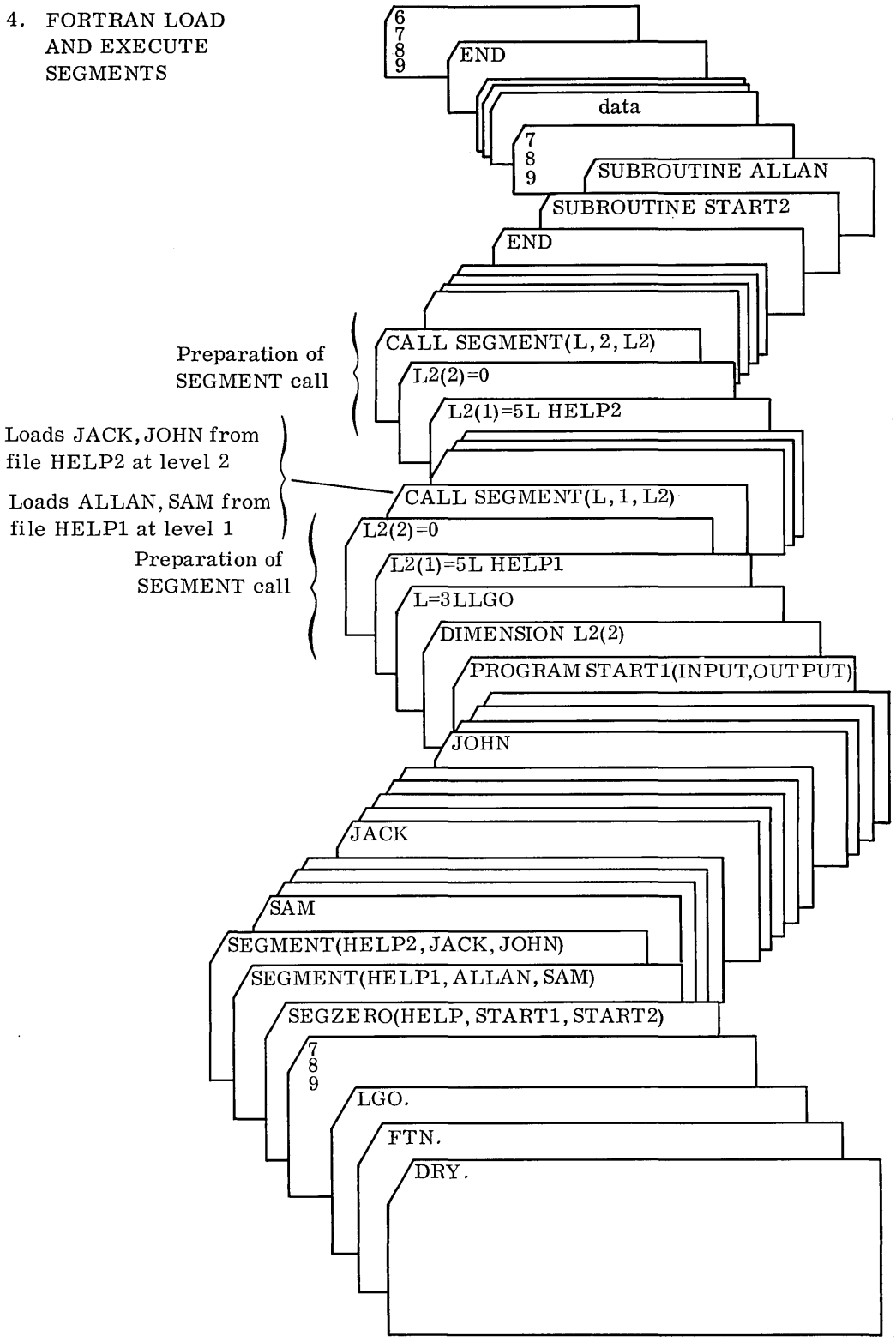
2. Compilation and Execution



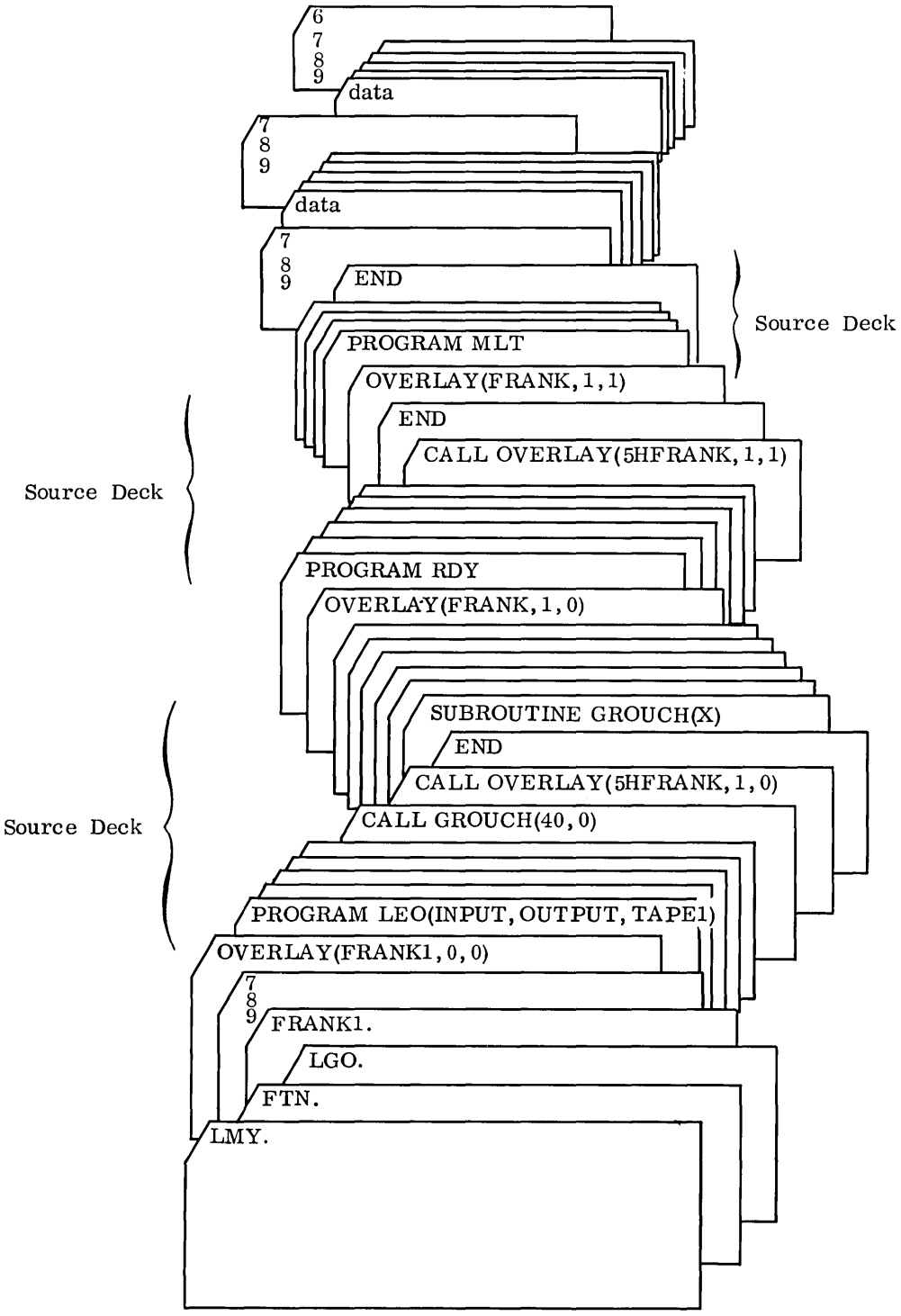
3. Compilation and Execution with Binary Subroutine



4. FORTRAN LOAD AND EXECUTE SEGMENTS



5. OVERLAY PREPARATION OF 0,0; 1,0; and 1,1



OBJECT TIME I/O

I

STRUCTURE OF FILES

A file is an ordered sequence of user logical records. Each type of input/output that a FORTRAN programmer can use has a user logical record definition:

FORMATTED I/O

READ f,k	PUNCH f,k	READ(u,f)	WRITE(u,f)
PRINT f,k	READ(u,f) k	WRITE(u,f) k	

For formatted I/O the user logical record (also referred to as a unit record) corresponds to a card image or a print line. User logical records may be a maximum of 150_{10} characters for input but only 137_{10} are transferred on output records. A user logical record corresponds to a tape block on S and L tapes†; on X tapes it always is 136_{10} characters.

UNFORMATTED I/O

READ (u) k	WRITE (u) k

When I/O is unformatted, the user logical record is the same as a SCOPE logical record on internal files or X†† magnetic tape files. On an S and L magnetic tape the physical representation of user logical records is the same as that on a SCOPE internal tape although there is no SCOPE-logical-record definition (i.e., on S- and L-style tapes each tape block will consist of a maximum of 5120 characters with a user logical record terminated by a tape block shorter than 5120 characters).

Since the physical representation of FORTRAN unformatted user logical records is the same on S and L tapes as that on SCOPE internal tapes, the files may be used interchangeably; a tape created as a SCOPE internal tape may be read as an S or L tape. Likewise, a tape created by a FORTRAN job as an S or L tape may be read as a SCOPE internal tape. Tapes written as X tapes must be read as X tapes.

Throughput of small user logical records can be increased if S magnetic tapes are used instead of SCOPE internal or L tapes. Non-stop tape motion can often be achieved when the buffer size is in excess of 2048_{10} words, which is four physical record units on magnetic tape.

BUFFER I/O

BUFFER IN (u,k) (A,B)	BUFFER OUT (u,k) (A,B)

†Stranger tape and Long record tape.
††External tape in SCOPE 2 format.

On SCOPE internal files (including tape files) and binary S magnetic tapes, the user logical record is represented as a SCOPE logical record. On a coded X tape, the user logical record will always consist of 14 words (136 characters on tape), and any attempt to write a record longer will result in a fatal diagnostic. On S and L magnetic tapes, the user logical record is defined to be one tape block, the information between two record gaps or between the load point and a record gap. On S magnetic tapes, 512 words is the maximum record length.

BUFFER I/O

BUFFEI (BUFFER IN)

Only one logical record is read each time BUFFEI is called. If the block length specified by the call is longer than the logical record, excess block locations will not be changed by the read. If the logical record is longer than the block, excess words in the logical record are passed over. The number of CM words transmitted to the program block may be obtained by referencing LENGTH.

After using a BUFFER IN (or BUFFER OUT) 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 check insures that requested data has 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-file without referencing the UNIT function, BUFFEI will abort the program with the diagnostic: *BUF IN**ENDFILE file name

If a read is attempted, when the last operation on the file was a write, BUFFEI will abort the program with the diagnostic: *BUF IN**LAST OP WRITE, file name

If the starting address for the block is greater than the terminal address, BUFFEI will abort the program with the diagnostic: *BUF IN***FWA.GT.LWA, file name

If an attempt is made to BUFFER IN from an undefined file (file not declared on the PROGRAM card), BUFFEI will abort the job with the diagnostic: *BUF IN**UNASSIGNED MEDIUM, file name

BUFFEO (BUFFER OUT)

One logical record is written each time the routine is called; record length is LWA-FWA+1.

A BUFFER OUT operation must be followed by a reference to the UNIT function. Since BUFFEO changes the buffer arguments for the file to point to the CM block specified in the call, calls to other routines involving the same file may not follow any buffer operation until the pointers have been restored by the UNIT function. If LWA is less than FWA, the program will be aborted and the following diagnostic will appear in the dayfile: *BUF OUT**FWA.GT.LWA, file name

The UNASSIGNED MEDIUM diagnostic is similar to that issued by BUFFEI.

Random Access Files (Mass Storage)

Two degrees of sophistication are available using the mass storage subroutines. It is possible to utilize the routines in a normal fashion having just one master index, or it is possible to have a master index and many sub-indexes. A file may have a name or a number index and is referenced in one of the following ways:

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

u is a logical unit number; ix is the first word address of the index in central memory; l is the index length; p indicates how the file is referenced; fwa is central memory address of first word of record; n is number of CM words to be transferred; i is record number or cell address of record name or number. (See Chapter 5, Mass Storage I/O.)

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 CM area specified in the call (the ix parameter).

The STINDX subroutine causes no transfer of data, it merely changes the file index in the FET to the base specified in the call. After calling STINDX it is necessary to call READMS or WRITMS to read in or create the new index. After making a call to STINDX, if the next operation on that file 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 word address is specified by the ix parameter in the call to STINDX) prior to calling WRITMS. The master index must be reset before termination of the job so that the correct index will be written on the file.

Upon termination of the job, the mass storage file is closed automatically by FORTRAN. At this time the index as specified in the FET is written as a record on the file.

Examples:

```
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 SUB-INDEX)
    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
```

```
2.  PROGRAM MS (TAPE5)
C PROGRAM FOR CREATING RANDOM FILE
      DIMENSION J(10),B(7),XYZ(20),ZXY(10),YXZ(50)
      DATA JOE,SAM,PETE,SUB1/3LJOE,3LSAM,4LPETE,4LSUB1/
      CALL OPENMS(5,J,10,1)
      CALL STINDX(5,B,7)
      DO 10 I=1,7
   10 B(I)=0.
C USE INDEX B
      CALL WRITMS(5,XYZ,20,JOE)
      CALL WRITMS(5,ZXY,10,SAM)
      CALL WRITMS(5,YXZ,50,PETE)
      CALL STINDX(5,J,10)
      CALL WRITMS(5,B,7,SUB1)
C WRITE OUT THE SUB-INDEX
      END

3.  PROGRAM MS (TAPE5)
C THIS MS FILE HAS NO SUB-INDEXES
      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
```

The execution-time routine END will close the file, causing the index at I to be rewritten on the file.

Status Checking

UNIT Function

The UNIT (i) function checks the status of a buffered operation (BUFFER IN or BUFFER OUT only) on logical unit i. The function returns values as follows:

-1	unit ready, no previous error
+0	previous read encountered an end-of-file
+1	parity error on previous read

Example:

```
IF(UNIT(i)) 12,14,16
```

Upon return from the UNIT function, control is transferred to the statement labeled 12, 14 or 16 if the value returned was -1, 0, or +1 respectively.

If the value returned is 0 or +1 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).

EOF Function

The EOF (i) function tests for end-of-file read (non-buffered) on unit i. The value zero is returned if no end-of-file was encountered on the previous read, or non-zero if end-of-file was encountered on unit i.

Example:

```
IF (EOF(i)) 10,20
```

If i designates the file named INPUT, control will return to statement 10 if the previous read encountered an end-of-file, or any 7/8/9 end-of-record card. Otherwise control will go to statement 20.

The user should make the EOF check after each READ operation to insure against possible input/output errors. If a READ on unit i is attempted and an EOF 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, EOF will always return a zero value. Only when an end-of-file is read will the end-of-file condition exist.

This function has no meaning when applied to a mass storage file. If the EOF function is called in reference to a MS file, a zero value is always returned.

IOCHEC Function

The IOCHEC (i) function tests for parity errors on non-buffered reads on unit i. The value zero is returned if no error occurs.

Example:

```
J = IOCHEC(i)
IF (J) 15,25
```

A value of zero is returned to J if no parity error occurs, and non-zero is returned otherwise. Control would then transfer to the statement labeled 25 or 15 respectively. If a parity error occurs, IOCHEC will clear the parity indicator before returning.

Parity errors are handled in the above fashion regardless of the type of the external device.

Only read parity errors are detected by the status checking functions. Write parity errors are detected and a message is written in the dayfile by the SCOPE system.

A parity error indication reveals parity error somewhere within the current logical record. For nonbuffered coded files, this does not necessarily mean the error occurred within the last record requested by the program because the I/O routines read a logical record ahead whenever possible.

Backspace/Rewind

If a BACKSPACE is requested on a coded file (except files created by the BUFFER OUT statement) the file is logically moved back one unit record. The backspace is attempted within the I/O buffer; if this is not possible, the external I/O device is repositioned.

Backspace on binary files and files created by BUFFER I/O statements reposition the external device so that the last logical record becomes the next logical record.

When a BACKSPACE (or REWIND) request follows a write operation on a file, an end-of-file is written followed by two backspaces (or by a rewind). Note that SCOPE may write trailer label information immediately following the end-of-file written by FORTRAN.

FORMAT Field Separators

Field descriptors are normally delimited by field separators; however, some exceptions are allowed. For example, the statement

```
10 FORMAT(F25.22F10.3)
```

would be interpreted as two descriptors, F25.22 and F10.3. Field separators should be used whenever ambiguity could result.

ENCODE/DECODE

Under SCOPE, a binary zero byte is used to terminate a unit record. When the DECODE processor encounters a zero character (6 bits of binary zeros), that character is interpreted as a blank. Conversion continues through n characters per record.

Whenever a record terminator (a slash or the right parenthesis if the list is not exhausted) is encountered in a FORMAT statement, the rest of the record is filled out with blanks (for ENCODE) or ignored (for DECODE), and conversion continues beginning with the next record. (The length of the record is specified by n in a DECODE (n,f,A)k 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)
10  FORMAT (10F10.4,/,6F10.4) is allowed
```

Labeled Files

Only files recorded on 1/2 inch magnetic tape may be labeled files.

When the PROGRAM line is compiled, FET's (File Environment Tables) are set up for each file declared. All the fields in the FET label information for a given file are set to zero with one exception; the reel number is set to 1. If the file has been declared as labeled on a REQUEST control card, SCOPE compares the label with the information in the FET when the file is opened. The information will not compare, and if the initial use of the file is for input SCOPE will allow the job to continue only after instructions are entered from the display console to do so. If the initial use of the file is for output SCOPE will write a default label on the tape and the job will continue.

In order for the FORTRAN programmer to compare label information or to create a standard label containing given information, an object time subroutine (LABEL) is provided to set the desired information into the FORTRAN prepared FET.

If the label information is properly set up, and subroutine LABEL is referenced prior to any other reference to the file, then when the file is opened 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)
```

where:

- u is the unit number
- fwa is the address of the first of four consecutive words containing the desired label information to be placed into the FET. The information must be in the mode and format discussed in Appendix C of the SCOPE 3.2 Reference Manual.

The four words beginning at fwa are transferred directly to words 10 through 13 of the FET for the file designated by u.

Carriage Control Characters

Character	Action Before Printing	Action After Printing
A	Space 1	Eject to top of next page†
B	Space 1	Skip to last line of page†
1	Eject to top of next page	No space†
2	Skip to last line on page	No space†
+	No space	No space
0 (zero)	Space 2	No space
- (minus)	Space 3	No space
blank	Space 1	

† The top of a page is indicated by a punch in channel 8 of the carriage control tape for the 501 printer and channel 1 for the 512 printer. The bottom of page is channel 7 in the 501 and 12 in the 512.

When the following characters are used for carriage control, no printing takes place. The remainder of the line will not be printed.

Q	Clear auto page eject
R	Select auto page eject
S	Clear 8 vertical lines per inch (512 printer)
T	Select 8 vertical lines per inch (512 printer)
PM (col 1-2)	Output remainder of line (up to 30 characters) on the B display and the dayfile and wait for the JANUS typein /OKuu. For files assigned to a printer, n.GO. must be typed to allow the operator to change form or carriage control tapes.
any other	See SCOPE Reference Manual.

Any pre-print skip operation of 1, 2 or 3 lines that follows a post skip operation will be reduced to 0, 1 or 2 lines.

The functions Q through T should be given at the top of a page. S and T can cause spacing to be different from the stated spacing if given in other positions on a page. Q and R will cause a page eject before the next line is printed.

Notes

Meaningful results are not guaranteed in the following circumstances:

1. Mixed mode files within a logical file.

2. Mixing buffer I/O statements and standard Read/Write statements on the same file (without a REWIND in between).

3. Requesting a LENGTH function on a buffer unit before requesting a UNIT function.

4. Two consecutive buffer I/O statements on the same file without the intervening execution of a UNIT function call.

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 R'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 R's in the output field if the internal data is out of range or I's if it is indefinite (as defined for 6400/6600 hardware).

Disposition of files at run termination:

1. All indexed files (randomly accessible files) are closed through SCOPE.
2. Output files are demarcated by FORTRAN with an end-of-file and are not rewound. No action is taken on input files.

SUBPROGRAM AND MEMORY STRUCTURE J

This appendix describes the arrangement of code and data within PROGRAM, SUBROUTINE and FUNCTION subprograms. It does not describe the arrangement of data within common blocks because this is specified by the programmer; however, their placement in memory is described.

SUBROUTINE and FUNCTION Structure

The code within procedure subprograms is arranged in the following blocks (relocation bases) in the given order.

START.	The code for the primary entry and the saving of A0.
VARDIM.	The address substitution code and the variable dimension initialization code.
ENTRY.	Either a full word of NO's or nothing.
CODE.	The code generated by compiling executable statements followed by parameter lists for external procedure references within the current procedure.
FORMAL PARAM-ETERS	One local block for each formal parameter in the order in which they appear on the subroutine header card, to hold tables used in address substitution for processing reference to dummy arguments.
DATA.	Storage for usage declared variables, format statements, constants and compiler generated temporaries.
DATA..	Storage for dimensioned local variables.
HOL.	Storage for Hollerith constants.

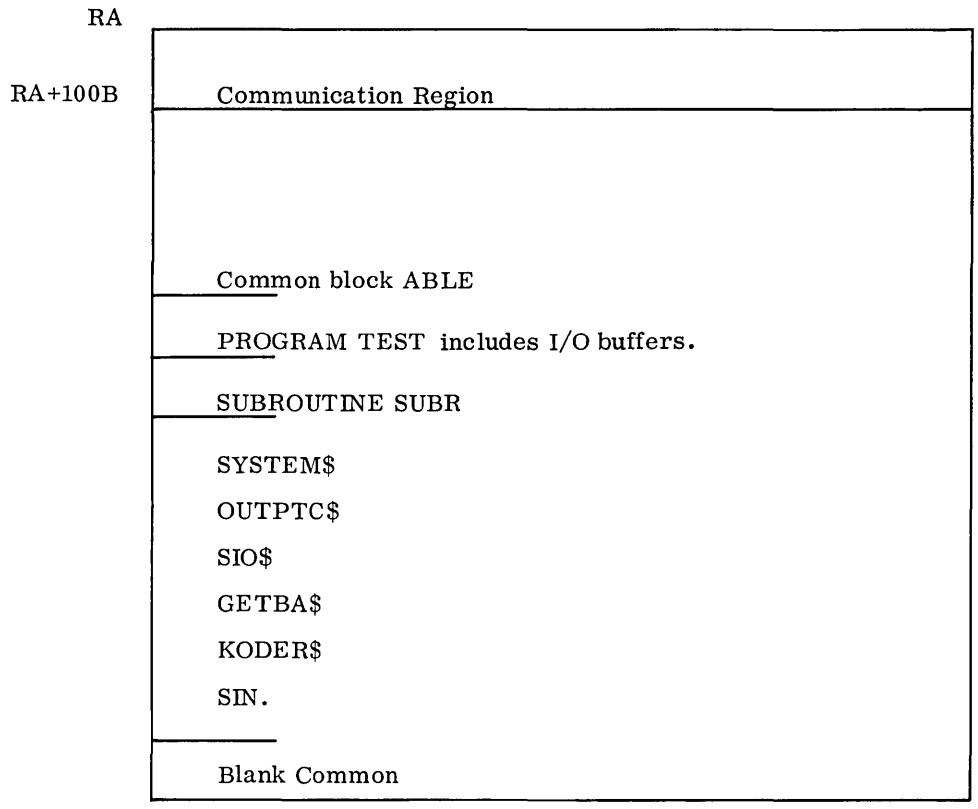
Main subprograms consist of the following blocks:

START.	The I/O buffers and a table of files specified in the PROGRAM card.
CODE.	The transfer address code plus the code specified for the CODE. block above.
DATA.	Storage for usage declared variables, format statements, constants and compiler generated temporaries.
DATA..	Storage for dimensioned local variables.
HOL.	Storage for Hollerith constants.

Memory Structure

Subprograms are loaded as encountered in the input file from RA+100B toward FL. Labeled common blocks are loaded prior to the subprogram in which they first occur. Library routines are loaded immediately after the last encountered subprogram and these are followed by blank common.

The following is a typical memory layout.



When a program is entered at an INTERCOM control point, INTERCOM associates INPUT and OUTPUT files of the program with the user's remote terminal device, and all references to these files are directed to the terminal. With calls to the CONDIS library subprogram, the user may specify other files to be associated with the terminal.

The user can associate any logical file in his program with a remote device, with the statement:

```
CALL CONNEC (lfn)
```

If a file is already connected, the request will be ignored. If the file has been used already, but not connected, this request will clear the file's buffer, write an end-of-file, and backspace over it before the connection is performed.

A file is disconnected by:

```
CALL DISCON (lfn)
```

This request will be ignored if the file is not connected. After a disconnect, the file is reassociated with its former device.

lfn File name parameter of the form:

tape logical unit number, 1 to 99

Hollerith constant in the format hLfilename

integer variable containing either of the above

Examples:

```
CALL CONNEC (3LEWT)
CALL DISCON (6)
K=5LINPUT
CALL DISCON (K)
J=12
CALL CONNEC (J)
```

Any files listed on the PROGRAM card may be connected or disconnected during program execution. An attempt to connect or disconnect an undefined file will result in a fatal execution time error, and the job will be terminated.

CONNEC and DISCON calls are ignored when programs are not executed through an INTERCOM control point.

Interactive input/output is supported only for formatted and NAMELIST reads and writes.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: 6400/6500/6600 FORTRAN Extended Reference Manual

PUBLICATION NO. 60176600 REVISION F

Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments:

FROM NAME: _____ POSITION: _____

BUSINESS ADDRESS: _____

_____

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
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE

► ►CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN, 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

Pub. No. 60176600

Litho in U.S.A.