CONTROL DATA
CORPORATION

CONTROL DATA®
CYBER 70 COMPUTER SYSTEMS
MODELS 72, 73, 74, 76
7600 COMPUTER SYSTEM
6000 COMPUTER SYSTEMS

FORTRAN EXTENDED REFERENCE MANUAL
MODELS 72, 73, 74 VERSION 4
MODEL 76 VERSION 2
7600 VERSION 2
6000 VERSION 4

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

REVISION RECORD

REVISION	DESCRIPTION
A	Original printing.
(10-22-71)	
Publication No. 60305600	

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
Software Documentation
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

PREFACE

This manual describes the FORTRAN Extended language (version 4.0) for the CONTROL DATA ® CYBER 70/ Models 72, 73 and 74, and 6200, 6400, 6500, 6600 and 6700 computers, and FORTRAN Extended (version 2.0) for the CONTROL DATA CYBER 70/Model 76, 7600, 7601-1 and 761X computers. It is assumed that the reader has knowledge of an existing FORTRAN language and the CONTROL DATA CYBER 70, 6000 Series or 7600 computer systems. FORTRAN Extended is designed to comply with American National Standards Institute FORTRAN language.

The FORTRAN compiler operates in conjunction with version 3.0 COMPASS assembly language processor under control of the 6000 SCOPE operating system (version 3.4), and 7000 SCOPE operating system (version 2.0). The FORTRAN compiler makes optimum use of the high speed execution characteristics of the CONTROL DATA CYBER 70, 6000 Series and 7600 computer systems. It utilizes the SCOPE operating system's multi-programming features to provide compilation and execution within a single job operation, as well as simultaneous compilation of several programs.

The following new features are included in FORTRAN Extended:

LEVEL statement

IMPLICIT statement

Hollerith strings in output lists

Expressions in output lists

Quote delimited Hollerith strings

Exclusive OR function

Messages on STOP and PAUSE statements

Line limit on output file at execution time

Syntax scan only during compilation

Program listings suppressed but reference map produced

Rewrite in place, mass storage

Multiple systems texts and local texts for intermixed COMPASS programs

This manual is in three parts. The reference section, Part 1, contains a full description of the FORTRAN Extended language.

Part 2 consists of a set of sample programs with input cards and output. Each program is preceded by a short introduction which explains some of the more difficult aspects of the language for the less experienced FORTRAN programmer.

Part 3 contains mainly systems information, although the applications programmer will be interested in the character set in section 1 and the compilation and execution diagnostics in section 2.

Throughout the manual, CONTROL DATA extensions to the FORTRAN language are indicated by blue type. Otherwise, FORTRAN Extended conforms to ANSI standards.

Information which applies only to the CONTROL DATA CYBER 70/Model 76 and 7600 computers is indicated by red type.

Information which applies only to the CONTROL DATA CYBER 70/Models 72, 73 and 74, and 6000 Series computers is indicated by green type.

CONTENTS

PART II

PART III

STATEMENT FORMS

The following symbols are used in the descriptions of FORTRAN Extended statements:

v	variable or array element
sn	statement label
iv	integer variable
name	symbolic name
u	input/output unit: 1- or 2-digit decimal integer constant integer variable with value of: 1-99 or display code file name
fn	format designator
iolist	input/output list

Other forms are defined individually in the following list of statements.

n	string of 1–5 octal digits
c . . . c	string of 1–70 characters

TYPE DECLARATION

	Page Numbers
COMMON $v_1, \ldots, v_n$	6-8
COMMON/blkname$_1$/$v_1, \ldots, v_n$ . . . /blkname$_n$/$v_1, \ldots, v_n$	6-8
COMMON// $v_1, \ldots, v_n$	6-8

blkname — symbolic name or 1 - 7 digits

// — blank common

DATA vlist$_1$/dlist$_1$/ , . . . , vlist$_n$/dlist$_n$/	6-21
DATA (var=dlist) , . . . , (var=dlist)	6-21

var — variable, array element, array name or implied DO list

vlist — list of array names, array elements, or variable names, separated by commas

dlist — one or more of the following forms separated by commas:

constant
(constant list)
rf*constant
rf*(constant list)
rf(constant list)

constant list — list of constants separated by commas

rf — integer constant. The constant or constant list is repeated the number of times indicated by rf

EQUIVALENCE ($v_1, \ldots, v_n$) , . . . , ($v_1, \ldots, v_n$)	6-11
LEVEL n, $a_1, \ldots, a_n$	6-17

n — unsigned integer 1, 2 or 3

a — variable, array element, array name

a	first word of data block to be transferred
b	last word of data block to be transferred
p	integer constant or integer variable. zero = even parity, nonzero = odd parity

	Page Numbers
NAMELIST/group name$_1$/$a_1, \ldots, a_n$/.../group name$_n$/$a_1, \ldots, a_n$	9-9
READ (u,group name)	9-10
WRITE (u,group name)	9-12

a_i — array names, array elements, or variables

group name — symbolic name identifying the group $a_1, \ldots, a_n$

INTERNAL TRANSFER OF DATA

ENCODE (c,fn,v) iolist	9-15
DECODE (c,fn,v) iolist	9-18

v — starting location of record. Variable or array name

c — length of record in characters. Unsigned integer constant or simple integer variable

FILE MANIPULATION

REWIND u	9-6
BACKSPACE u	9-6
ENDFILE u	9-6

FORMAT SPECIFICATION

sn FORMAT ($fs_1, \ldots, fs_n$)	10-5

fs_i — one or more field specifications separated by commas and/or grouped by parentheses

Page Number

OVERLAYS

CALL OVERLAY (fname,i,j,recall,k)		12-5
i	primary overlay number	
j	secondary overlay number	
recall	if 6HRECALL is specified, the overlay is not reloaded if it is already in memory	
k	L format Hollerith constant: name of library from which overlay is to be loaded	
	any other non-zero value: overlay loaded from global library set	
OVERLAY (fname,i,j,Cn)		12-4
i	primary overlay number, octal	
j	secondary overlay number, octal	
Cn	n is a 6-digit octal number indicating start of load relative to blank common	

DEBUG

C$ DEBUG		13-24
C$ DEBUG (name$_1$, . . . , name$_n$)		13-24
C$ AREA bounds$_1$, . . . , bounds$_n$ C$ DEBUG	within program unit	13-27
C$ AREA/name$_1$/bounds$_1$, . . . , bounds$_n$, . . . , /name$_n$/bounds$_1$, . . . , bounds$_n$ C$ DEBUG (name$_1$, . . . , name$_n$) or C$ DEBUG	external debug deck	13-27

bounds	(n_1,n_2)	n_1 initial line position n_2 terminal line position
	(n_3)	n_3 single line position to be debugged
	(n_1,*)	n_1 initial line position * last line of program
	(*,n_2)	* first line of program n_2 terminal line position
	(*,*)	* first line of program * last line of program

		Page Numbers
C$	ARRAYS $(a_1, \ldots, a_n)$	13-4
C$	ARRAYS	13-4
	a_i array names	
C$	CALLS $(s_1, \ldots, s_n)$	13-6
C$	CALLS	13-6
	s_i subroutine names	
C$	FUNCS $(a_1, \ldots, a_n)$	13-8
C$	FUNCS	13-8
	f_i function name	
C$	GOTOS	13-15
C$	NOGO	13-18
C$	STORES $(c_1, \ldots, c_n)$	13-11

c_i variable name

variable name .relational operator. constant

variable name .relational operator. variable name

variable name .checking operator.

checking operators:

RANGE	out of range
INDEF	indefinite
VALID	out of range or indefinite

C$	TRACE (lv)	13-16
C$	TRACE	13-16

lv level number:

0	tracing outside DO loops
n	tracing up to and including level n in DO nest

C$	OFF	13-28
CS	OFF $(x_1, \ldots, x_n)$	13-28

x_i any debug option

CODING FORTRAN STATEMENTS

A FORTRAN program contains executable and non-executable statements. Executable statements specify action the program is to take, and non-executable statements describe characteristics of operands, statement functions, arrangement of data, and format of data.

The FORTRAN source program is written on the coding form illustrated in figure 1. Each line on the coding form represents an 80-column card. The source language character set (section 1, part 3) is used to code statements.

THE FORTRAN CHARACTER SET

Alphabetic	A to Z	Alphanumeric
Numeric	0 to 9	

Special

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

In addition, any of the SCOPE set may be used in Hollerith constants and in comments. Blanks are not significant except in Hollerith fields.

FORTRAN STATEMENTS

Column 1	C or $ or * indicates comment line
Columns 1-2	C$ indicates debug statement
Columns 1-5	Statement label
Column 6	Any character other than blank or zero denotes continuation; does not apply to comment cards. A debug continuation card must contain C$ in columns 1-2.
Columns 7-72	Statement
Columns 73-80	Identification field, not processed by compiler

CONTINUATION

Statements are coded in columns 7-72; if a statement is longer than 66 columns, it may be continued on as many as 19 lines. A character other than blank or zero in column 6 indicates a continuation line. Column 1 can contain any character other than C *, or $; columns 2, 3, 4 and 5 may contain any character. Any statement except a comment, END or OVERLAY may be continued.

STATEMENT SEPARATOR

Several short statements may be written on one line if each is separated by the special character $. The statement following the $ sign is treated as a separate statement. For example:

```
 7
 ACUM=24.$I=0 $ IDIFF=1970-1626
```

is the same as

```
 7
 ACUM = 24.
 I = 0
 IDIFF = 1970-1626
```

$ may be used with all statements except FORMAT, OVERLAY, or debug statements. The statement following $ must not be labeled; the information following $ is treated exactly as if it were punched into column 7 on a succeeding card.

STATEMENT LABELS

Columns 1-5 of the first line of a statement may be used for the statement label. All executable statements (except END) may be labeled. Statements referenced by other statements must be labeled. Statement labels are integers 1-99999, and they may appear in any order. Leading zeros and leading or embedded blanks are not significant. Each statement label must be unique to the program unit in which it appears. In figure 1, statement labels are 4, 1, 2, and 3.

CONTROL DATA CORPORATION

FORTRAN CODING FORM

PROGRAM PASCAL	NAME
ROUTINE	DATE / PAGE OF

```
      PROGRAM PASCAL (OUTPUT)
      INTEGER L(11)
      DATA L(11)/1/
C
      PRINT 4, (I,I=1,11)
    4 FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME./20X,3HN-/
     $11I5)
C
      DO 2 I=1,10
      K=11-I
      L(K)=1
      DO 1 J=K,10
    1 L(J)=L(J)+L(J+1)
    2 PRINT 3, (L(J),J=K,11)
    3 FORMAT(11I5)
      STOP
      END
```

Figure 1. Program PASCAL

COMMENTS

In column 1 a C, *, or $ indicates a comment line. Comments do not affect the program; they can be written in column 2 to 80 and can be placed anywhere within the program. If a comment occupies more than one line, each line must begin with C, *, or $ in column 1. The continuation character in column 6 does not apply to comment cards. Comments can appear between continuation cards.

COLUMNS 73-80

Any information may appear in columns 73-80 as they are not part of the statement. Entries in these columns are copied to the source program listing, but they are not processed by the compiler. They are generally used to order the punched cards in a deck.

BLANK CARDS

If blank cards are used to separate statements, they will produce a blank line on the source listing. A line following a blank card is treated as a new statement; therefore continuation cards must not follow blank cards.

A blank card should not follow the END statement. If it does an informative diagnostic is printed.

DATA CARDS

No restrictions are imposed on the format of data cards read by the source program. Data can be written in columns 1-80. Columns 73-80 are not ignored on data cards.

LANGUAGE ELEMENTS I-2

CONSTANTS AND VARIABLES

CONSTANTS

A constant is a fixed quantity. The seven types of constants are: integer, real, double precision, complex, octal, Hollerith, and logical.

INTEGER CONSTANT

$\boxed{n_1 n_2 \cdots\cdots n_m}$

$1 \leq m \leq 18$ decimal digits

Examples:

```
237     -74     +136772     0     -0024
```

An integer constant is a string of 1-18 decimal digits written without a decimal point. It may be positive, negative or zero. If the integer is positive, the plus sign may be omitted; if it is negative, the minus sign must be present. An integer constant must not contain a comma. The range of an integer constant is $-2^{59}-1$ to $2^{59}-1$ ($2^{59}-1$ = 576 460 752 303 423 487).

Examples of invalid integer constants:

`46.`	(decimal point not allowed)
`23A`	(letter not allowed)
`7,200`	(comma not allowed)

When the integer constant is used as a subscript, or as the index in a DO statement or an implied DO, the maximum value is $2^{17}-2$ ($2^{17}-2$ = 131 070), and minimum is 1.

Integers used in multiplication and division are truncated to 48 bits. The result of integer multiplication or division will be less than $2^{48}-1$. If the result is larger than $2^{48}-1$, ($2^{48}-1$ = 281 474 976 710 655) high order bits will be lost. No diagnostic is provided. The resultant maximum value of conversion from real to integer or integer to real is $2^{48}-1$. If the value exceeds $2^{48}-1$, high order bits are lost; no diagnostic is provided. For addition and subtraction, the full 60-bit word is used.

REAL CONSTANT

n.n	n.	n.nE±s	.nE±s	n.E±s	nE±s

n — Coefficient ≤ 15 decimal digits

E±s — Exponent

s — Base 10 scale factor

A real constant consists of a string of decimal digits written with a decimal point or an exponent, or both. Commas are not allowed. If positive, a plus sign is optional.

The range of a real constant is 10^{-293} to 10^{+322}; if this range is exceeded, a diagnostic is printed. Precision is approximately 14 decimal digits, and the constant is stored internally in one computer word.

Examples:

```
7.5    -3.22    +4000.    23798.14    .5    - .72    42.E1    700.E-2
```

Examples of invalid real constants:

`3,50.`	(comma not allowed)
`2.5A`	(letter not allowed)

A real constant may be followed by a decimal exponent, written as the letter E and an integer constant indicating the power of ten by which the number is to be multiplied. The field following the letter E may be zero, but it must not be omitted. The sign may be omitted if the exponent is positive, but it must be present if the exponent is negative. The range of the integer exponent is -308 through +337.

Examples:

`42.E1`	($42. \times 10^1 = 420.$)
`.00028E+5`	($.00028 \times 10^5 = 28.$)
`6.205E12`	($6.205 \times 10^{12} = 6205000000000.$)
`8.0E+6`	($8. \times 10^6 = 8000000.$)
`700.E-2`	($700. \times 10^{-2} = 7.$)
`7E20`	($7. \times 10^{20} = 70\ 000\ 000\ 000\ 000\ 000\ 0000.$)

Example of invalid real constants:

`7.2E3.4`	exponent not an integer

DOUBLE PRECISION CONSTANT

n.nD±s .nD±s n.D±s nD±s

n	Coefficient
D±s	Exponent
s	Base 10 scale factor

Double precision constants are written in the same way as real constants except the exponent is specified by the letter D instead of E. Double precision values are represented internally by two computer words, giving extra precision. A double precision constant is accurate to approximately 29 decimal digits.

Examples:

`5.834D2`	$(5.834 \times 10^2 = 583.4)$
`14.D-5`	$(14. \times 10^{-5} = .00014)$
`9.2D03`	$(9.2 \times 10^3 = 9200.)$
`-7.D2`	$(-7. \times 10^2 = -700.)$
`3120D4`	$(3120. \times 10^4 = 31200000.)$

Examples of invalid double precision constants:

`7.2D`	exponent missing
`D5`	exponent alone not ailowed
`2,1.3D2`	comma illegal
`3.14159265358979323846264338327 9`	D missing

COMPLEX CONSTANT

$\boxed{\text{(r1, r2)}}$

r1	Real part
r2	Imaginary part

Each part has the same range as a real constant.

Complex constants are written as a pair of real constants separated by a comma and enclosed in parentheses.

FORTRAN Coding	Complex Number	
`(1., 7.54)`	1. + 7.54i	$i = \sqrt{-1}$
`(-2.1E1, 3.24)`	-21. + 3.24i	
`(4.0, 5.0)`	4.0 + 5.0i	
`(0., -1.)`	0.0 - 1.0i	

The first constant represents the real part of the complex number, and the second constant represents the imaginary part. The parentheses are part of the constant and must always appear. Either constant may be preceded by a plus or minus sign. Complex values are represented internally by two consecutive computer words.

Both parts of complex constants must be real; they may not be integer.

Examples of invalid complex constants:

`(275, 3.24)`	275 is an integer
`(12.7D-4 16.1)`	comma missing and double precision not allowed
`4.7E+2,1.942`	parentheses missing
`(0,0)`	0 is an integer

Real constants which form the complex constant may range from 10^{-293} to 10^{+322}.

OCTAL CONSTANT †

$\boxed{n_1 \ldots n_m B}$

n ≤ m ≤ 20 octal digits

An octal constant consists of 1 to 20 octal digits suffixed with the letter B.

Examples:

```
777777B

52525252B

500127345B
```

Invalid octal constants:

`892777B`	8 and 9 are non-octal digits
`7700000000777775252525 2B`	exceeds 20 digits
`07766`	O not allowed

An octal constant must not exceed 20 digits nor contain a non-octal digit. If it does, a fatal compiler diagnostic is printed. When fewer than 20 octal digits are specified, the digits are right justified and zero filled. Octal constants can be used anywhere integer constants can be used, except: they cannot be used as statement labels or statement label references, in a FORMAT statement, or as the character count when a Hollerith constant is specified.

They can be used in DO statements, expressions, and DATA statements, and as DIMENSION specifications.

Examples:

`BAT = (I*5252B) .OR. JAY`	masking expression
`J = MAXO (I,1000B,J,K+40B)`	octal constant used as parameter in function
`NAME = I .AND. 77700000B`	masking expression
`J = (5252B + N)/K`	arithmetic expression
`DIMENSION BUF(1000B)`	dimension specification

When an octal constant is used in an expression, it assumes the type of the dominant operand of the expression (Table 3-1, section 3).

†Blue type indicates non-ANSI statements.

HOLLERITH CONSTANTS

nHf	nLf
nRf	≠f≠

n	Unsigned decimal integer representing number of characters in string. Must be greater than zero, and not more than 10 when used in an expression.
f	String of characters
≠≠	String delimiters
H	Left justified with blank fill
L	Left justified with binary zero fill
R	Right justified with binary zero fill

```
    5 7
      PROGRAM HOLL (OUTPUT)
      A = 6HABCDEF
      B = 6LABCDEF
      C = 6RABCDEF
      D = ≠ABCDEF≠
      PRINT 1, A,A,B,B,C,C,D,D
    1 FORMAT (O24,A15)
      STOP
      END
```

Stored Internally:	Display Code:	
01020304050655555555	ABCDEF	A
01020304050600000000	ABCDEF	B
00000000010203040506	ABCDEF	C
01020304050655555555	ABCDEF	D

A Hollerith constant consists of an unsigned decimal integer, the letter H, and a string of characters. For example:

```
5HLABEL
```

The integer represents the number of characters in the string. Spaces are significant in a Hollerith constant:

```
18HTHIS IS A CONSTANT

7HTHE END

19HRESULT NUMBER THREE
```

`I = (+5HABCDE)` is a valid statement; (+5HABCDE) is an expression and the + sign is an operator.

nHf

Hollerith constants may be used in arithmetic expressions, DATA and FORMAT statements, as arguments in subprogram calls, and as list items in an output list of an input/output statement. If a Hollerith constant is used as an operand in an arithmetic operation, an informative diagnostic is given. In an expression or a DATA statement, a Hollerith constant is limited to 10 characters. In a FORMAT statement or as an actual argument to a subprogram, the length of the Hollerith string is limited to 150 characters.

A Hollerith string delimited by the paired symbols ≠ ≠ can be used anywhere the H form of the Hollerith constant can be used. For example,

```
IF(V.EQ.≠YES≠) Y=Y+1.

PRINT 1, ≠ SQRT = ≠, SQRT(4.)

PRINT 2, ≠ TEST PASSED ≠

INTEGER LINE(7), N1THRU9
LOGICAL NEWPAGE
IF (NEWPAGE) LINE(7) = ≠ PAGE 0 ≠ + N1  THRU 9
```

```
    PROGRAM FL(OUTPUT)
    PRINT 1, ≠ FIELD LENGTH = ≠, IGETFL(I)
1   FORMAT (2A10,I6)
    END
```

The symbol ≠ can be represented within the string by two successive ≠ symbols.

When the number of characters in a Hollerith constant is less than 10, the computer word is left justified with blank fill. If it is more than 10, but not a multiple of 10, only the last computer word is left justified with blank fill.

Examples:

```
      7
      READ 1,NAME
    1 FORMAT (A7)
      IF(NAME .EQ. 4HJOAN) GO TO 20
```

```
      7
      WRITE (6,1000)
 1000 FORMAT (1X, 73H NO COUNTRY THAT HAS BEEN THOROUGHLY EXPLORED IS
     S INFESTED WITH DRAGONS.)
```

nRf and nLf

A Hollerith constant of the form R or L is limited to 10 characters and cannot be used in a FORMAT statement.

LOGICAL CONSTANTS

A logical constant takes the forms:

.TRUE. or .T. representing the value true

.FALSE or .F. representing the value false

The decimal points are part of the constant and must appear.

Examples:

```
    5 7
      PROGRAM LOGIC(INPUT,OUTPUT,TAPE5=INPUT)
      LOGICAL MALE,PHD,SINGLE,ACCEPT
      INTEGER AGE
      PRINT 20
   20 FORMAT (*1                    LIST OF ELIGIBLE CANDIDATES*)
    3 READ (5,1) LNAME,FNAME,MALE,PHD,SINGLE,AGE
    1 FORMAT (2A10,3L5,I2)
      IF (EOF(5))6,4
 4    ACCEPT = MALE .AND. PHD .AND. SINGLE .AND. (AGE .GT. 25 .AND.
     S     AGE .LT. 45)
      IF (ACCEPT) PRINT 2,LNAME,FNAME,AGE
    2 FORMAT (1H0,2A10,3X,I2)
      GO TO 3
 6    STOP
      END
```

(An explanation of this example appears in part 2.)

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

VARIABLE NAMES

Unless otherwise stated, the term variable applies to both Large Core Memory (LCM) and Small Core Memory (SCM) variables. †

A variable represents a quantity whose value may vary; this value may change repeatedly during program execution. Variables are identified by a symbolic name of 1-7 alphanumeric characters, the first of which must be alphabetic. A variable is associated with a storage location; and whenever the variable is used, it assumes the value currently in that location. The five types of variables are: logical, integer, real, double precision and complex.

The type of a variable is implied by its first character if it is not defined explicitly with a type declaration (section 5). If type is not declared, a variable is type integer if the first character of the symbolic name is I, J, K, L, M or N, and if no IMPLICIT statement appears in that program unit.

†Red type applies to 7600 computer and CYBER 70 Model 76.

Examples:

```
IFORM   JINX2   KODE   NEXT23   M
```

A variable not defined in a type declaration is type real if the first character of the symbolic name is any letter other than I, J, K, L, M, N, and if no IMPLICIT statement appears in that program unit.

Examples:

```
RESULT   ASUM   A73   BOX
```

Implied Typing of Variables

A-H, O-Z	Real
I-N	Integer

INTEGER VARIABLE

An integer variable name must be 1-7 alphanumeric characters; the first letter must be I, J, K, L, M, or N if the type has not been defined explicitly.

The value range is $-2^{59}-1$ to $2^{59}-1$. When an integer variable is used as a subscript or as the index in a DO statement, the maximum value is $2^{17}-2$. The resultant absolute value of conversion from integer to real, integer multiplication, integer division, or input/output under the I format specification must be less than $2^{48}-1$. If this value is exceeded, high order bits will be lost. The resultant absolute value of integer addition or subtraction must be less than $2^{59}-1$.

Examples:

```
ITEM1   NSUM   JSUM   N72   J   K2SO4
```

REAL VARIABLES

A real variable name must be 1-7 alphanumeric characters of which the first must be any letter other than I, J, K, L, M, or N if the type has not been defined explicitly.

The value range is 10^{-293} to 10^{+322}, with approximately 14 significant digits.

Examples:

```
AVAR   SUM3   RESULT   TOTAL2   BETA   XXXX
```

DOUBLE PRECISION VARIABLES

Double precision variable names must be defined explicitly by a type declaration. The value of a double precision variable may range from 10^{-293} to 10^{+322}, with approximately 29 significant digits.

Example:

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

COMPLEX VARIABLES

Complex variables must be defined explicitly by a type declaration. A complex variable occupies two words in storage. Each word contains a number in real variable format, and each number can range from 10^{-293} to 10^{+322}.

Example:

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

LOGICAL VARIABLES

Logical variables must be defined explicitly by a type declaration. A logical variable has the value true or false. A logical variable with a positive zero value is false; any other value is true.

Example:

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

OCTAL AND HOLLERITH DATA

Octal and Hollerith data can be entered or used in any type variable. When an octal or Hollerith constant is used in an arithmetic operation, it needs no conversion. If the constant 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`	For addition, octal constants are treated as 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, which is real, is converted to integer before being stored in KLM.

ARRAYS

A FORTRAN array is a set of elements identified by a single name. A particular element in the array may be referenced by its position in the array. Arrays may have one, two, or three dimensions; the array name and dimensions must be declared in a DIMENSION, COMMON or type declaration.

Example:

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA(X,12)
      CALL IOTA(Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (*1ARRAY X = *,12F6.0,5X,*ARRAY Y = *6F6.0)
      STOP
      END
```

The number of elements in an array is the product of the dimensions. For example, STOR(3,7) contains 21 elements, STOR(6,6,3) contains 108. The number of subscripts must not exceed the number specified in the array declaration. For example, a one dimensional array A(I) cannot be referred to as A(I,J) and a two dimensional array A(I,J) cannot be referred to as A(I,J,K). Such references would produce a diagnostic.

The number of dimensions in the array is indicated by the number of subscripts in the declaration.

`DIMENSION STOR(6)`	declares a one-dimensional array of six elements
`REAL STOR(3,7)`	declares a two-dimensional array of three rows and seven columns
`LOGICAL STOR(6,6,3)`	declares a three-dimensional array of six rows, six columns and three planes

Each element in the array is referred to by the array name followed by a set of expressions in parentheses, called subscripts. Subscripts indicate the position of the element in the array.

Example:

The array N consists of six values in the order: 10, 55, 11, 72, 91, 7

N(1)	value 10
N(2)	value 55
N(3)	value 11
N(4)	value 72
N(5)	value 91
N(6)	value 7

The entire array may be referenced by the unsubscripted array name when it is 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.

Example:

The two-dimensional array TABLE (4,3) has four rows and three columns.

	Column 1	**Column 2**	**Column 3**
Row 1	44	10	105
Row 2	72	20	200
Row 3	3	11	30
Row 4	91	76	714

To refer to the number in row two, column three write TABLE(2,3).

TABLE(3,3) = 30 TABLE(1,1) = 44 TABLE(4,1) = 91

TABLE(4,4) would be outside the bounds of the array and results may be unpredictable.

Zero and negative subscripts are not allowed. If the number of subscripts in a reference is less than the declared dimensions, the compiler assumes missing subscripts have a value of one.

For example, in an array 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(I,J)

A(I) implies A(I,1)

A implies A(1,1)†

and for A(I)

A implies A(1)†

For example, in a three-dimensional array NEXT when only one subscript is shown, the remaining subscripts are assumed to be one.

	Plane 1			Plane 2			Plane 3		
	Col 1	Col 2	Col 3	Col 1	Col 2	Col 3	Col 1	Col 2	Col 3
Row 1	3	7	4	22	51	7	2	1	552
Row 2	7	8	9	0	98	6	77	60	3
Row 3	0	33	2	3	207	99	85	100	8

the single subscript NEXT (3) represents NEXT (3,1,1)

NEXT (3,2) represents NEXT (3,2,1)

NEXT (2,2) represents NEXT (2,2,1)

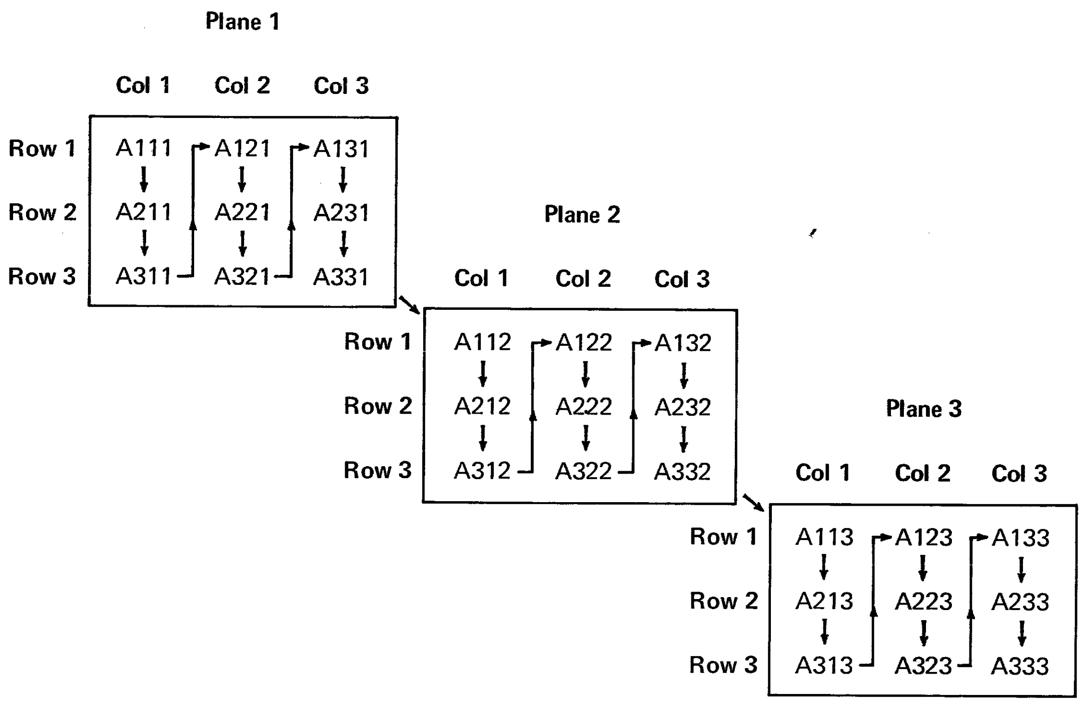
†Except in input/output lists, as arguments to functions or subroutines, and DATA statements.

ARRAY STRUCTURE

Arrays are stored in ascending locations; the value of the first subscript increases most rapidly, and the value of the last increases least rapidly.

Example:

In an array declared as A(3,3,3), the elements of the array are stored by columns in ascending locations.



The array is stored in linear sequence as follows:

Element		Location Relative to first Element
A(1,1,1)	stored in	0
A(2,1,1)		1
A(3,1,1)		2
A(1,2,1)		3
A(2,2,1)		4
A(3,2,1)		5
A(1,3,1)		6
A(2,3,1)		7
A(3,3,1)		8
A(1,1,2)		9
A(2,1,2)		10
A(3,1,2)		11
A(1,2,2)		12
A(2,2,2)		13
A(3,2,2)		14
A(1,3,2)		15
A(2,3,2)		16
A(3,3,2)		17
A(1,1,3)		18
A(2,1,3)		19
A(3,1,3)		20
A(1,2,3)		21
A(2,2,3)		22
A(3,2,3)		23
A(1,3,3)		24
A(2,3,3)		25
A(3,3,3)	stored in	26

To find the location of an element in the linear sequence of storage locations the following method can be used:

Number of Dimensions	Array Dimension	Subscript	Location of Element Relative to Starting Location
1	ALPHA(K)	ALPHA(k)	(k-1XE)
2	ALPHA(K,M)	ALPHA(k,m)	(k-1+KX(m-1))XE
3	ALPHA(K,M,N)	ALPHA(k,m,n)	(k-1+KX(m-1+MX(n-1)))XE

Figure 2-1. Array Element Location

K, M, and N are dimensions of the array.

k,m, and n are the actual subscript values of the array.

1 is subtracted from each subscript value because the subscript starts with 1, not 0.

E is length of the element. For real, logical, and integer arrays, E = 1. For complex and double precision arrays, E = 2.

Examples:

	Subscript	Location of Element Relative to Starting Location
INTEGER ALPHA (3)	ALPHA(2)	(2-1 = 1)
REAL ALPHA (3,3)	ALPHA(3,1)	(3-1+3X(1-1))X1 = 2
REAL ALPHA (3,3,3)	ALPHA(3,2,1)	(3-1+3X(2-1)+3X3X(1-1))X1 = 5

A single subscript may be used for an array with multiple dimensions.

The amount of storage allocated to arrays is discussed under DIMENSION declarations in Section 5.

SUBSCRIPTS

A subscript can be any valid arithmetic expression. If the value of the expression is not integer, it is truncated to integer.

The value of the subscript must be greater than zero and less than or equal to the maximum specified in the array specification statement, or the reference will be outside the array. If the reference is outside the bounds of the array, results are unpredictable.

Examples:

Valid subscripts:

```
A(I,K)
B(I+2,J-3,6*K+2)
LAST(6)
ARAYD(1,3,2)
STRING(3*K*ITEM+3)
```

Invalid subscripts:

`ATLAS(0)`	zero subscript not allowed
`D(1 .GE. K)`	relational or logical expression illegal
`Z14(-4)`	negative subscript not allowed

EXPRESSIONS I-3

FORTRAN expressions are arithmetic, masking, logical and relational. Arithmetic and masking expressions yield numeric values, and logical and relational expressions yield truth values.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of unsigned constants, variables, and function references separated by operators and parentheses. For example,

(A-B)*F+C/D**E is a valid arithmetic expression

FORTRAN arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

An arithmetic expression may consist of a single constant, variable, or function reference. 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	

All operations must be specified explicitly. For example, to multiply two variables A and B, the expression A*B must be used. AB, (A)(B), or A.B will not result in multiplication.

Expression	Value of
`3.78542`	Real constant 3.78542
`A(2*J)`	Array element A (2*J)
`BILL`	Variable BILL
`SQRT(5.0)`	$\sqrt{5.}$
`A+B`	Sum of the values A and B
`C*D/E`	Product of C times D divided by E
`J**I`	Value of J raised to the power of I
`(200 - 50)*2`	300

EVALUATION OF EXPRESSIONS

The precedence of operators for the evaluation of expressions is shown below:

**	(exponentiation)
/ *	(division or multiplication)
+ −	(addition or subtraction)
.GT. .GE. .LT. .LE. .EQ. .NE.	(relationals)
.NOT.	(logical)
.AND.	(logical)
.OR.	(logical

Unary addition or subtraction are treated as operations on an implied zero. For example, +2 is treated as 0+2, -3 is treated as 0-3.

Expressions are evaluated from left to right with the precedence of the operators and parentheses controlling the sequence of operation (the deepest nested parenthetical subexpression is evaluated first).

However, any function references and exponentiation operations not evaluated inline are evaluated prior to other operations.

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

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 arguments or subscripts.

The evaluation of an expression having 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 (section 4, part 3)

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

If the error traceback option is selected on the FTN control card (section 11), the first three conditions will produce informative diagnostics during execution. If the traceback option is not selected, a mode error message is printed (section 4, part 3).

Two operators must not be used together. A*-B and Z/+X are not allowed. However, a unary + or - can be separated from another operator in an expression by using parentheses. For example,

`A*(-B)` and `Z/(+X)`	Valid expressions
`B*-A` and `X/-Y*Z`	Invalid expressions

Each left parenthesis must have a corresponding right parenthesis.

Example:

`(F + (X * Y)`	Incorrect, right parenthesis missing
`(F + (X * Y))`	Correct

Examples:

In the expression A-B*C

B is multiplied by C, and the product is subtracted from A.

The expression A/B-C*D**E is evaluated as:

D is raised to the power of E.

A is divided by B.

C is multiplied by the result of D**E.

The product of C*D**E is subtracted from the quotient of A divided by B.

The expression -A**C is evaluated as 0-A**C; A is first raised to the power of C and the result is then subtracted from zero.

An expression containing operators of equal precedence is evaluated from left to right.

```
A/B/C
```

A is divided by B, and the quotient is divided by C. (A/B)/C is an equivalent expression.

The expression A**B**C is, in effect, ((A**B)**C).

Dividing an integer by another integer yields a truncated result; 11/3 produces the result 3. Therefore, when an integer expression is evaluated from left to right, J/K*I may give a different result than I*J/K.

Example:

```
I = 4      J = 3      K = 2

J/K*I          I*J/K

3/2*4 = 4      4*3/2 = 6
```

An integer divided by an integer of larger magnitude yields the result 0.

Example:

```
N = 24     M = 27     K = 2

N/M*K

24/27*2 = 0
```

Examples of valid expressions:

```
A

3.14159

B + 16.427

(XBAR +(B(I,J+I,K) /3.0))

-(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))
```

TYPE OF ARITHMETIC EXPRESSIONS

An arithmetic expression may be of type integer, real, double precision, or complex. The order of dominance from highest to lowest is as follows:

Complex

Double Precision

Real

Integer

Table 3-1. Mixed Type Arithmetic Expressions

1st operand† \ 2nd operand	Hollerith	Integer	Real	Double Precision	Complex	Octal
Hollerith	Integer	Integer	Real	Double Precision	Complex	Integer
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

† Operators are + - * /

When an expression contains operands of different types, type conversion takes place during evaluation. Before each operation is performed, operands are converted to the type of the dominant operand. Thus the type of the value of the expression is determined by the dominant operand. For example, in the expression A*B-I/J, A is multiplied by B, I is divided by J as integer, converted to real, and subtracted from the result of A multiplied by B.

EXPONENTIATION

In exponentiation, the following types of base and exponent are permitted:

Base	Power
Integer	Integer, Real, Double Precision, Complex
Real	Integer, Real, Double Precision, Complex
Double Precision	Integer, Real, Double Precision, Complex
Complex	Integer

The exponent is evaluated from left to right. The expression A**B**C is, in effect, ((A**B)**C)

In an expression of the form A**B the type of the result is determined as follows:

Type of A	Type of B	Type of Result of A**B
Integer	Integer Real Double Complex	Integer Real Double Complex
Real	Integer Real Double Complex	Real Real Double Complex
Double	Integer Real Double Complex	Double Double Double Complex
Complex	Integer	Complex

The expression -2**2 is equivalent to 0-2**2. An exponent may be an expression. The following examples are all acceptable:

```
B**2.

B**N

B**(2*N-1)

(A+B)**(-J)
```

A negative exponent must be enclosed in parentheses:

```
A**(-B)

NSUM**(-J)
```

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

RELATIONAL EXPRESSIONS

$\boxed{a_1 \quad op \quad a_2}$

a_1, a_2	Arithmetic or masking expression
op	Relational operator

A relational expression is constructed from arithmetic or masking expressions and relational operators. Arithmetic expressions may be type integer, real, double precision, or complex. The relational operators are:

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

The enclosing decimal points are part of the operator and must be present.

Two expressions separated by a relational operator constitute a basic logical element. The value of this element is either true or false. If the expressions satisfy the relation specified by the operator, the value is true; if not, it is false. For example:

```
X+Y .GT. 5.3
```

If X+Y is greater than 5.3 the value of the expression is true. If X+Y is less than or equal to 5.3 the value of the expression is false.

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

Relational operands may be of type integer, real, double precision, or complex, but not logical. With the exception of the relational operators .EQ. and .NE., only the real part of complex operands are used in evaluation.

Examples:

`J.LT.ITEM`	
`580.2 .GT. VAR`	
`B .GE. (2.7,5.9E3)`	real part of complex number is used in evaluation
`E.EQ..5`	
`(I) .EQ. (J(K))`	
`C.LT. 1.5D4`	most significant part of double precision number is used in evaluation

EVALUATION OF RELATIONAL EXPRESSIONS

Relational expressions are evaluated according to the rules governing arithmetic expressions. Each expression is evaluated and compared with zero to determine the truth value. For example, the expression p.EQ.q is equivalent to the question, does p - q = 0? q is subtracted from p and the result is tested for zero. If the difference is zero or minus zero the relation is true. Otherwise, the relation is false.

If p is 0 and q is -0 the relation is true.

Expressions are evaluated from left to right. Parentheses enclosing an operand do not affect evaluation; for example, the following relational expressions are equivalent:

```
A.GT.B

A.GT.(B)

(A).GT.B

(A).GT.(B)
```

Examples:

```
REAL A                      AMT .LT. (1.,6.55)
A.GT.720
                            DOUBLE PRECISION BILL, PAY
INTEGER I,J                 BILL .LT. PAY
I.EQ.J(K)
                            A+B.GE.Z**2
(I).EQ.(N*J)
                            300.+B.EQ.A-Z
B.LE.3.754
                            .5+2. .GT. .8+AMNT
Z.LT.35.3D+5
```

Examples of invalid expressions:

`A .GT. 720 .LE. 900`	2 relational operators must not appear in a relational expression
`B .LE. 3.754 .EQ. C`	

LOGICAL EXPRESSIONS

$\boxed{L_1 \text{ op } L_2 \text{ op } L_3 \text{ op} \ldots L_n}$

$L_1 \ldots L_n$	logical operand or relational expression
op	logical operator

A logical expression is a sequence of logical constants, logical variables, logical array elements, or relational expressions separated by logical operators and possibly parentheses. After evaluation, a logical expression has the value true or false.

Logical operators:

.NOT. or .N.	logical negation
.AND. or .A.	logical multiplication
.OR. or .O.	inclusive OR

The enclosing decimal points are part of the operator and must be present.

The logical operators are defined as follows (p and q represent LOGICAL expressions):

.NOT.p	If p is true, .NOT.p has the value false. If p is false, .NOT.p has the value true.
p.AND.q	If p and q are both true, p.AND.q has the value true. Otherwise, false.
p.OR.q	If either p or q, or both, are true then p.OR.q has the value true. If both p and q are false, then p.OR.q has the value false.

Truth Table

p	q	p .AND. q	p .OR. q	.NOT. p
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

If precedence is not established explicitly by parentheses, operations are executed in the following order:

.NOT.

.AND.

.OR.

Example:

```
    FROGRAM LOGIC(INFUT,OLTPUT,TAPE5=INPUT)
    LOGICAL MALE,FHD,SINGLE,ACCEPT
    INTEGER AGE
    FRINT 20
 20 FORMAT (*1                   LIST OF ELIGIBLE CANDIDATES*)
  3 READ (5,1) LNAME,FNAME,MALE,PHD,SINGLE,AGE
  1 FORMAT (2A10,3L5,I2)
    IF (EOF(5))6,4
 4  ACCEPT = MALE .AND. PHD .AND. SINGLE .AND. (AGE .GT. 25 .AND.
   S     AGE .LT. 45)
    IF (ACCEPT) PRINT 2,LNAME,FNAME,AGE
  2 FORMAT (1H0,2A10,3X,I2)
    GO TO 3
6   STOP
    END
```

Data Cards:

```
 RALPH    ERICSON   T     T     T     20
JOHN S.   SLIGHT    T     T     T     26
MILDRED   MINSTER   F     T     T     41
 JUSTIN   BROWN     T     T     T     30
 JAMES    SMITH     T     F     T     27
```

Output:

```
                 LIST OF ELIGIBLE CANDIDATES

JOHN S.   SLIGHT         26

 JUSTIN   BROWN          30
```

The operator .NOT. which indicates logical negation appears in the form:

.NOT. p

.NOT. may appear in combination with .AND. or .OR. only as follows (p and q are logical expressions):

p .AND..NOT. q

p .OR..NOT. q

p .AND.(.NOT. q)

p .OR.(.NOT. q)

.NOT. may appear adjacent to itself only in the form .NOT.(.NOT.(.NOT.p))

Two logical operators may appear in sequence only in the forms .OR..NOT. and .AND..NOT.

Valid Logical Expressions:

LOGICAL M,L

.NOT.L

.NOT. (X .GT. Y)

X .GT. Y .AND..NOT.Z

(L) .AND. M

Invalid Logical Expressions:

P,Q, and R are type logical

.AND. P	.AND. must be preceded by a logical expression
.OR. R	.OR. must be preceded by a logical expression
P.AND..OR.R	.AND. always must be separated from .OR. by a logical expression

Examples:

A, X, B, C, J, L, and K are type logical.

Expression	Aternative Form
`A .AND. .NOT. X`	`A .A. .N. X`
`.NOT.B`	`.N.B`
`A.AND.C`	`A .A.C`
`J.OR.L.OR.K`	`J.O.L.O.K`

Examples:

`B-C` ≤ `A` ≤ `B+C` is written as `B-C .LE. A .AND. A .LE. B+C`
`FICA` > `176.` and `PAYNB = 5889.` is written `FICA .GT. 176. .AND. PAYNB .EQ. 5889.`

MASKING EXPRESSIONS

Masking expressions are similar to logical expressions, but the elements of the masking expression are of any type variable, constant, or expression other than logical.

Examples:

```
J .AND. N             .NOT. (B)

.NOT. 55              KAY .OR. 63
```

Masking operators are identical in appearance to logical operators but meanings differ. In order of dominance from highest to lowest, they are:

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

The enclosing decimal points are part of the operator and must be present. Masking operators are distinguished from logical operators by non-logical operands.

Examples:

Expression	Alternative Form
`B .OR. D`	`B .O. D`
`A .AND. .NOT. C`	`A .A. .N. C`
`BILL .AND. BOB`	`BILL .A. BOB`
`I .OR. J .OR. K .OR. N`	`I .O. J .O. K .O. N`
`(.NOT. (.NOT.(.NOT. A .OR. B)))`	`(.N.(.N.(.N. A.OR. B)))`

The operands may be any type variable, constant, or expression (other than logical).

Examples:

`TAX .AND. INT`	
`.NOT. 55`	
`734 .OR. 82`	
`A .AND. 77B`	Extract the low order 6 bits of A
`B .OR. C`	Logical sum of the contents of B and C
`M .AND. .NOT. 77B`	Clear the low order 6 bits of M.

In masking operations operands are considered to have no type. If either operand is type COMPLEX, operations are performed only on the real part. If the operand is DOUBLE PRECISION only the most significant word is used. The operation is performed bit-by-bit on the entire 60-bit word. For simplicity, only 10 bits are shown in the following examples. Masking operations are performed as follows:

J = 0101011101 and I 1100110101

J .AND. I

The bit-by-bit logical product is formed

J 0101011101

I 1100110101

0100010101   Result after masking

J .OR. I

The bit-by-bit logical sum is formed

J 0101011101

I 1100110101

1101111101   Result after masking

.NOT. Complement the operand

.NOT. I

I 1100110101

0011001010

Result after masking

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

masking expression .AND. .NOT. masking expression

masking expression .OR. .NOT. masking expression

masking expression .AND. (.NOT. masking expression)

masking expression .OR. (.NOT. masking expression)

If an expression contains masking operators of equal precedence, the expression is evaluated from left to right.

A .AND. B .AND. C

A .AND. B is evaluated before B .AND. C

Using the following numbers:

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

Masking operations produce the following octal results:

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

Invalid example:

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

masking expression must not contain logical operand

Example:

```
      PROGRAM MASK (INPUT,OUTPUT)
1     FORMAT (1H1,5X,4HNAME,///)
      PRINT 1
2     FORMAT (3A10,I1)
3     READ 2,LNAME,FNAME,ISTATE,KSTOP
      IF(KSTOP.EQ.1)STOP
C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD
      IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.777700000000000
     K00000B))  GO TO 3
11    FORMAT(5X,2A10)
10    PRINT 11,LNAME,FNAME
      GO TO 3
      END
```

ASSIGNMENT STATEMENTS

An assignment statement evaluates an expression and assigns this value to a variable or array element. The statement is written as follows:

v = expression

v is a variable or an array element

The meaning of the equals sign differs from the conventional mathematical notation. It means replace the variable on the left with the value of the expression on the right. For example, the assignment statement A=B+C replaces the current value of the variable A with the value of B+C.

ARITHMETIC ASSIGNMENT STATEMENTS

v = arithmetic expression

Replace the current value of v with the value of the arithmetic expression. The variable or array element can be any type other than logical.

Examples:

`A=A+1`	replace the value of A with the value of A+1
`N=J-100*20`	replace N with the value of J-100*20
`WAGE=PAY-TAX`	replace WAGE with the value of PAY less TAX
`VAR=VALUE+(7/4)*32`	replace the value of VAR with the value of VALUE+(7/4)*32
`B(4)=B(1)+B(2)`	replace the value of B(4) with the value of B(1)+B(2)

If the type of the variable on the left of the equals sign differs from that of the expression on the right, type conversion takes place. The expression is evaluated, converted to the type of the variable on the left, and then replaces the current value of the variable. The type of an evaluated arithmetic expression is determined by the type of the dominant operand. Below, the types are ranked in order of dominance from highest to lowest:

Complex

Double Precision

Real

Integer

In the following tables, if high order bits are lost by truncation during conversion, no diagnostic is given.

CONVERSION TO INTEGER

	Value Assigned	Example	Value of IFORM After Evaluation
Integer = Integer	Value of integer expression replaces v.	IFORM = 10/2	5
Integer = Real	Value of real expression, truncated to 48-bit integer, replaces v.	IFORM = 2.5*2+3.2	8
Integer = Double Precision	Value of double precision expression, truncated to 48-bit integer, replaces v.	IFORM = 3141.593D3	3141593
Integer = Complex	Value of real part of complex expression truncated to 48-bit integer, replaces v.	IFORM = (2.5,3.0) + (1.0,2.0)	3

CONVERSION TO REAL

	Value Assigned	Example	Value of AFORM After Evaluation
Real = Integer	Value of integer expression, truncated to 48 bits, is converted to real and replaces v.	AFORM = 200 + 300	500.0
Real = Real	Value of real expression replaces v.	AFORM = 2.5 + 7.2	9.7
Real = Double Precision	Value of most significant part of expression replaces v.	AFORM = 3421.D – 04	.3421
Real = Complex	Value of real part of complex expression replaces v.	AFORM = (9.2,1.1) – (2.1,5.0)	7.1

CONVERSION TO DOUBLE PRECISION

	Value Assigned	Example	Value of SUM After Evaluation
Double Precision = Integer	Value of integer expression, truncated to 48 bits, is converted to real and replaces most significant part. Least significant part set to 0.	SUM = 7*5	35.D0
Double Precision = Real	Value of real expression replaces most significant part; least significant part is set to 0.	SUM = 7.5*2	15.D0

CONVERSION TO DOUBLE PRECISION (CONTINUED)

	Value Assigned	Example	Value of SUM After Evaluation
Double Precision = Double Precision	Value of double precision expression replaces v.	SUM = 7.322D2 - 32.D -1	7.29D2
Double Precision = Complex	Value of real part of complex expression replaces v. Least significant word is set to 0.	SUM = (3.2,7.6) + (5.5,1.0)	8.7D0

CONVERSION TO COMPLEX

	Value Assigned	Example	Value of AFORM After Evaluation
Complex = Integer	Value of integer expression, truncated to 48 bits, is converted to real, and replaces real part of v. Imaginary part is set to 0.	AFORM = 2 + 3	(5.0,0.0)
Complex = Real	Value of real expression replaces real part of v. Imaginary part set to 0.	AFORM = 2.3 + 7.2	(9.5,0.0)
Complex = Double Precision	Most significant part of double precision expression replaces real part of v. Imaginary part set to 0.	AFORM = 20D0 + 4.4D1	(64.0,0.0)
Complex = Complex	Value of complex expression replaces variable.	AFORM = (3.4,1.1) + (7.3,4.6)	(10.7,5.7)

LOGICAL ASSIGNMENT

Logical variable or array element = Logical or relational expression

Replace the current value of the logical variable or array element with the value of the expression.

Examples:

```
LOGICAL LOG2
I = 1
LOG2 = I .EQ.0
```

LOG2 is assigned the value .FALSE. because $I \neq 0$

```
LOGICAL NSUM,VAR
BIG = 200.
VAR = .TRUE.
NSUM = BIG .GT. 200. .AND. VAR
```

NSUM is assigned the value .FALSE.

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

MASKING ASSIGNMENT

v = masking expression

Replace the value of v with the value of the masking expression. v can be any type other than logical. No type conversion takes place during replacement. If the type is double precision or complex, the value of the expression is assigned to the first word of the variable; and the least significant or imaginary part set to zero.

Examples:

```
B = D .AND. Z .OR. X
SUM = (1.0,2.0) .OR. (7.0,7.0)
NAME = INK .OR. JAY .AND. NEXT
J(3) = N .AND. I
A = (B.EQ.C) .OR. Z
```

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

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

MULTIPLE ASSIGNMENT

$\boxed{v_1 = v_2 = \ldots v_n = \text{expression}}$

Replace the value of several variables or array elements with the value of the expression. For example, X = Y = Z= (10+2)/SUM(1) is equivalent to the following statements:

```
Z = (10 + 2)/SUM(1)

Y = Z

X = Y
```

The value of the expression is converted to the type of the variable or array element during each replacement.

Examples:

```
NSUM = BSUM = ISUM = TOTAL = 10.5 - 3.2
```

1. TOTAL is assigned the value 7.3
2. ISUM is assigned the value 7
3. BSUM is assigned the value 7.0
4. NSUM is assigned the value 7

Multiple assignment is legal in all types of assignment statements.

CONTROL STATEMENTS

FORTRAN statements are executed sequentially. However, the normal sequence may be altered with control statements.

ASSIGN	PAUSE
GO TO	STOP
IF	END
DO	RETURN
CONTINUE	

Control may be transferred to an executable statement only; a transfer to a non-executable statement results in a fatal diagnostic. Compilation continues, but the program is not executable unless it is compiled in debug mode.

Statements are identified by an integer, 1-99999. Leading zeros are ignored. Each statement number must be unique in the program or subprogram in which it appears.

In the following control statements:

sn = statement label

iv = integer variable

GO TO STATEMENT

The three GO TO statements are: unconditional, computed, and assigned.

UNCONDITIONAL GO TO

```
      7
      GO TO sn
```

Control transfers to the statement labeled sn.

Example:

```
   10 A=B+Z
  100 B=X+Y
      IF(A-B)20,20,30
   20 Z=A
      GO TO 10 <-------- Transfers control to statement 10
   30 Z=B
      STOP
      END
```

COMPUTED GO TO

```
      GO TO (sn1,sn2 ,..., snm),iv
```

```
      7
      GO TO (sn1,sn2,...,snm),expression
```

The comma separating the statement label list and the variable or expression is optional. This statement causes a transfer to one of the statement labels in parentheses, depending on the value of the variable. The variable, iv, can be replaced by an expression. The value of the expression is truncated and converted to integer if necessary, and used in place of iv.

Example:

```
GO TO(10,20,30,20),L

GO TO(10,20,30,20)L
```

The next statement executed will be:

10 if L = 1

20 if L = 2

30 if L = 3

20 if L = 4

The variable must not be specified by an ASSIGN statement. If it is specified by an ASSIGN statement, the object code is incorrect, but no compilation error message is issued.

If the value of the expression is less than 1, or larger than the number of statement numbers in parentheses, the transfer of control is undefined and a fatal error results. For example, execution of the following computed GO TO statement will cause a fatal error.

```
M=4
GO TO (100,200,300),M
```

Less than 4 numbers are specified in the list of statement numbers; therefore, the next statement to be executed is undefined.

Examples:

```
   K=2
   GO TO(100,150,300)K       statement 150 will be executed next

   K=2
   X=4.6
   .
   .
   .
   GO TO(10,110,11,12,13),X/K        control transfers to statement 110 since the integer
                                     part of the expression X/K equals 2

   L = 7
   GO TO(35,45,20,10)L-5     statement 45 will be executed next.
   .
   .
   .
35 Z=R+X
   .
   .
   .
45 A=X+Y
   .
   .
   .
20 B=CAT**2
   .
   .
   .
10 ANS=RES+ERROR
```

ASSIGN STATEMENT

```
      7
      ASSIGN sn TO iv
```

The value of iv is a statement label to which control may transfer. This statement is used in conjunction with the assigned GO TO statement. sn must be the label of an executable statement in the same program unit as the ASSIGN statement.

Example:

```
ASSIGN 10 TO LSWITCH
GO TO LSWITCH(5,10,15,20)
```

control transfers to statement 10

Once the integer variable, iv, is used in an ASSIGN statement, it must not be referenced in any statement, other than an assigned GO TO, until it has been redefined.

ASSIGNED GO TO

GO TO iv, (sn_1 ,. . . ,sn_m)

7

GO TO iv (sn_1,. . . , sn_m)

Example:

```
   ASSIGN 50 TO CHOICE
10 GO TO CHOICE,(20,30,40,50)
   .
   .
   .
30 CAT=ZERO+HAT
   .
   .
   .
40 CAT=10.1-3.
   .
   .
   .
50 CAT=25.2+7.3
```

statement 50 is executed immediately after statement 10

This statement transfers control to the statement label last assigned to the variable. The assignment must take place in a previously executed ASSIGN statement.

The comma after iv is optional. Omitting the list of statement labels (sn_1,...,sn_m) causes a fatal error. If the value of iv is defined by a statement other than an ASSIGN statement, the results are unpredictable. (A transfer is made to the absolute memory address represented by the low order 18 bits of iv.)

The ASSIGN statement assigns to the variable one of the statement labels specified in parentheses.

Example:

```
GO TO NAPA,(5,15,25)
```

If 5 is assigned to NAPA, statement 5 is executed next, if 15 is assigned to NAPA, statement 15 is executed next, if 25 is assigned to NAPA, statement 25 is executed next.

ARITHMETIC IF

THREE BRANCH

```
      7
      IF (arithmetic or masking expression) sn1,sn2,sn3
```

expression < 0 branch to sn_1

expression = 0 branch to sn_2

expression > 0 branch to sn_3

This statement transfers control to sn_1 if the value of the arithmetic or masking expression is less than zero, sn_2 if it is equal to zero, or sn_3 if it is greater than zero. Zero is defined as a word containing all bits set to zero or all bits set one (+0 or -0).

Example:

```
      PROGRAM IF (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      READ (5,100) I,J,K,N
  100 FORMAT (10X,4I4)
      IF(I-N) 3,4,6
    3 ISUM=J+K
    6 CALL ERROR1
      PRINT 2, ISUM
    2 FORMAT (I10)
    4 STOP
      END
```

If the type of the evaluated expression is complex, only the real part is tested.

ARITHMETIC IF

TWO BRANCH

```
      7
      IF (expression) sn1,sn2
```

expression is a masking or arithmetic expression

This statement transfers control to sn_1 if the value of the expression is not equal to 0, and to sn_2 if it is equal to 0.

Example:

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

LOGICAL IF

```
      7
      IF (logical or relational expression) stat
```

stat is any executable statement other than DO, END or a logical IF.

If the expression is true, stat is executed. If the expression is false, the statement immediately following the IF statement is executed.

`.FALSE.` = +0
`.TRUE.` is any value other than +0

Examples:

```
   IF (P.AND.Q) RES=7.2
50 TEMP=ANS*Z
```

If P and Q are both true, the value of the variable RES is replaced by 7.2. Otherwise, the value of RES is unchanged. In either case, statement 50 is executed.

```
   IF (A.LE. 2.5) CASH=150.
70 B=A+C-TEMP
```

If A is less than or equal to 2.5, the value of CASH is replaced by 150. If A is greater than 2.5 CASH remains unchanged.

```
   IF (A.LT.B) CALL SUB1
20 ZETA=TEMP+RES4
```

If A is less than B, the subroutine SUB1 is called. Upon return from this subroutine, statement 20 is executed. If A is greater than or equal to B, statement 20 is executed, and SUB1 is not called.

LOGICAL IF

TWO BRANCH

```
7
IF (logical or relational expression) sn1,sn2
```

If the value of the expression is true, sn_1 is executed. If the value of the expression is false, sn_2 is executed.

Example:

```
IF(K.EQ.100)60,70
```

If K is equal to 100, statement 60 is executed; otherwise statement 70 is executed.

DO STATEMENT

```
DO sn iv=m1,m2,m3
```

```
7
DO sn iv=m1,m2
```

iv is a non-subscripted integer variable called the index

m_1, m_2, m_3, the indexing parameters, may be unsigned integer or octal constants or simple integer variables with positive values no larger than 2^{17}-2. If m_3 is not specified, it is assigned the value 1. If the indexing parameters exceed 2^{17}-2 (or 10 digits) with or without leading zeros, the performance of the loop is unspecified.

The DO statement is used to execute repeatedly a section of program up to and including the statement labeled sn. sn must be an executable statement in the same program unit as the DO statement. If m_1 exceeds m_2 on initial entry to the loop, the loop is executed once; and control passes to the statement following sn.

m_1 is the initial value assigned to iv; m_2 is the limit value, and m_3 is the amount added to the initial value each time the DO loop is executed. When the value of iv exceeds m_2, the DO loop is completed; and control passes to the statement following sn. At execution, m_1, m_2, and m_3 must be greater than zero. The range of each DO loop contains all executable statements between and including the first executable statement after the DO and the terminal statement identified by sn. An extended range is a transfer of control out of the range of a DO loop followed by a transfer back into the same DO loop.

The control variable and the parameters m_1, m_2, and m_3 may not be redefined during execution of the immediate or extended range of the DO. When parameters are redefined during execution, the results are unpredictable. An informative diagnostic is issued for redefinition during an immediate range.

The last statement in a DO loop must not be an arithmetic IF or GO TO statement, a two branch logical IF, a RETURN, END, STOP, PAUSE or another DO statement, or a logical IF containing any of the preceding statements.

Examples:

```
      DO 10 I=1,11,3
      IF(ALIST(I)-ALIST(I+1))15,10,10
   15 ITEMP=ALIST(I)
   10 ALIST(I)=ALIST(I+1)
  300 WRITE(6,200)ALIST
```

The statements following DO up to and including statement 10 are executed 4 times. The DO loop is executed with I equal to 1,4,7,10. Statement 300 is then executed.

```
      K=3
      J=5
      DO 100 I=J,K
      RACK=2.-3.5+ANT(I)
  100 CONTINUE
```

The DO loop would be executed once only (with I = 5) because J is larger than K.

```
      DO 10 I=1,5
      CAT=BOX+D
   10 IF (X.GT.B.AND.X.LT.H)Z=EQUATE
    6 A=ZERO+EXTRA
```

Statement 10 is executed five times whether or not Z = EQUATE is executed. Statement 6 is executed only after the DO loop is satisfied.

After the last execution of the DO loop, control passes to the statement following sn, and the DO is satisfied. When the DO is satisfied, the index variable iv becomes undefined. A transfer out of the range of a DO loop is permissible at any time. When such a transfer occurs, the index variable iv remains defined as its most recent value in the loop.

Example:

```
      IVAR = 9
      .
      .
      .
      DO 20 I = 1,200
      IF (I-IVAR) 20,10,10
   20 CONTINUE
   10 IN = 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 IVAR. The value of the integer variable, IN, becomes 9.

LOOP TRANSFER

The range of a DO statement may include other DO statements providing the range of each inner DO is entirely within the range of the containing DO statement. The last statement of an inner DO loop must be either the same as the last statement of the outer DO loop or occur before it.

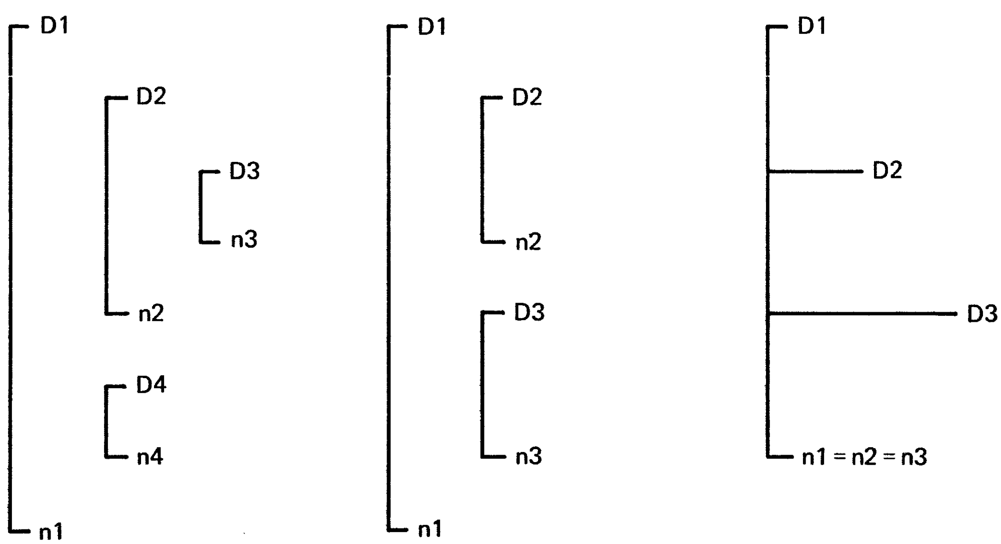
If more than one DO loop has the same terminal statement, a transfer to that statement may be made only from within the range (or extended range) of the innermost DO. When a DO loop contains another DO loop, the grouping is called a DO nest. DO loops may be nested to 50 levels.

Example:

```
      DIMENSION A(5,4,4), B(4,4)
      DO 2 I = 1,4
      DO 2 J = 1,4
      DO 1 K = 1,5
    1 A(K,J,I) = 0.0
    2 B(J,I) = 0.0
```

Examples:

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



The preceding diagrams could be coded as follows:

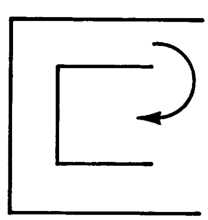
```
      DO 1 I=1,10,2
      .
      .
      DO 2 J=1,5
      .
      .
      .
      DO 3 K=2,8
      .
      .
      .
    3 CONTINUE
      .
      .
      .
    2 CONTINUE
      .
      .
      .
      DO 4 L=1,3
      .
    4 CONTINUE
      .
    1 CONTINUE
```

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

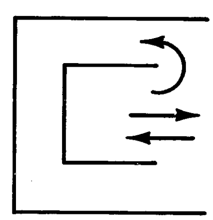
```
      DO 5 I=1,5
      DO 5 J=I,10
      DO 5 K=J,15
      .
      .
      .
    5 A = B*C
```

A DO loop may be entered only through the DO statement. Once the DO statement has been executed, and before the loop is satisfied, control may be transferred out of the range and then transferred back into the range of the DO.

A transfer from the range of an outer DO into the range of an inner DO loop is not allowed. However, a transfer out of the range of an inner DO into the range of an outer DO is allowed because such a transfer is within the range of the outer DO loop.

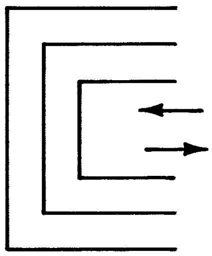


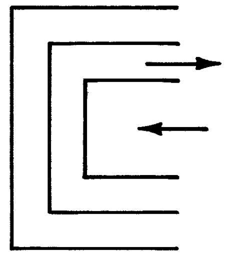
Illegal



Legal

The use of, and return from, a subprogram within a DO loop is permitted. A transfer back into the range of an innermost DO loop is allowed if a transfer has been made from that **same** loop.





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:

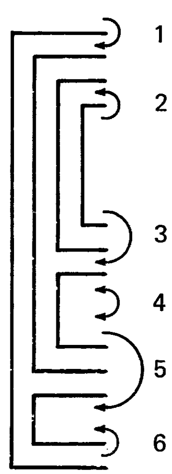
```
      DO 10 J=1,50
      DO 10 I=1,50
      DO 10  M=1,100
      .
      .
      .
      GO TO 10
      .
      .
      .
   10 CONTINUE
```

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
```

In the following illustration, transfers 2, 3, and 4 are acceptable; 1, 5, and 6 are not.



CONTINUE

```
    5 7
   sn CONTINUE
```

Example:

```
      DO 10 I = 1,11
      IF (A(I)-A(I+1)20,10,10
   20 ITEMPP = A(I)
      A (I) = A (I+1)
   10 CONTINUE
```

CONTINUE is a statement that may be placed anywhere in the source program without affecting the sequence of execution. It is used most frequently as the last statement in the range of a DO loop to avoid ending the loop with an illegal statement. The CONTINUE statement should contain a statement label in columns 1-5. If it does not, it serves no purpose; and an informative diagnostic is provided.

```
      DO 20 I=1,20
    1 IF (X(I) - Y(I))2,20,20
    2 X(I)=X(I)+1.0
      Y(I)=Y(I)-2.0
      GO TO 1
   20 CONTINUE
```

The use of the CONTINUE statement avoids ending the DO loop with the statement GO TO 1.

PAUSE

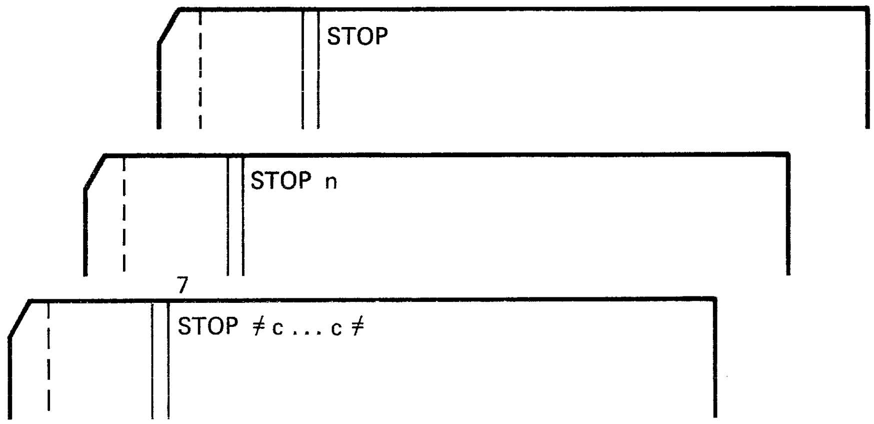
```
      PAUSE

      PAUSE n

      7
      PAUSE ≠ c...c ≠
```

n is a string of 1-5 octal digits.

c...c is a string of 1-70 characters.

When a PAUSE statement is encountered during execution, the program halts and PAUSE n, or c...c, 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.

STOP

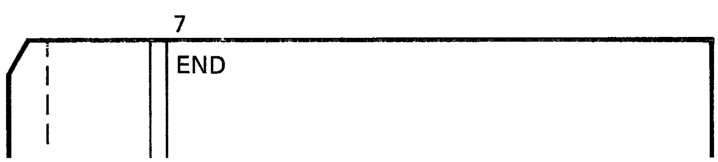


n is a string of 1-5 octal digits.

c...c is a string of 1-70 characters.

When a STOP statement is encountered during execution, STOP n, or STOP c...c, is displayed in the dayfile, the program terminates and control returns to the operating system. If n is omitted, blanks are implied. A program unit may contain more than one STOP statement.

END



The END line terminates compilation of a program unit. This line should be the last statement in a program or subprogram.If an END line is omitted and a SCOPE end of record or end of file immediately follows the last source program statement, an informative diagnostic is printed.

If an END statement is executed in a subprogram, control returns to the calling program. When an END statement is encountered in the main program of an overlay, control returns to the statement following the CALL OVERLAY statement which initialized loading and execution of the overlay.

The END line can follow a statement separator ($), and can be continued. No blank cards, nor blank continuation cards, should follow the card containing the final character D. If they do, an informative diagnostic is printed.

```
      PROGRAM RLIST (INPUT, OUTPUT)
      READ 5,X,Y,Z
    5 FORMAT (3F10.4)
      RESULT = X-Y+Z
      PRINT 100,  RESULT
  100 FORMAT (11H1 RESULT IS ,F7.3)
      END
```

RETURN

```
      RETURN
```

```
      RETURN i
```

i is a dummy argument which appears in the RETURNS list

RETURN returns control from a subprogram to the calling program. Control returns to the next executable statement following the CALL. In function subprograms, a RETURN statement returns control to the statement containing the function reference. A subprogram may contain more than one RETURN statement. A RETURN statement in a main program has the same effect as an END line, and an informative message is issued during compilation.

Example:

```
      A = SUBFUN (D,E)            FUNCTION SUBFUN(X,Y)
   10 DO 200 I = 1,5              SUBFUN = X/Y
      .                           RETURN
      .                           END
      .
```

RETURN i can appear only in a SUBROUTINE subprogram with a RETURNS list. (A RETURN i in a FUNCTION subprogram causes a fatal error at compilation time.) The statement labels in the RETURNS list in the CALL statement correspond to the dummy statement labels in the SUBROUTINE statement in the SUBROUTINE subprogram. When a SUBROUTINE subprogram is called, the actual statement labels replace the dummy statement labels. Execution of RETURN i returns control to the statement label corresponding to i in the RETURNS list.

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
```

Program MAIN passes statement labels 101,102,103 and 104 to subroutine XCOMP to replace the dummy RETURNS arguments A1,A2,A3 and A4. If RETURN A1 is reached in the subroutine, a return is made to statement 101; if A2 is reached, a return is made to statement 102, A3 to 103, and A4 to 104.

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
```

Example:

```
      FUNCTION Y(X)
      IF (X.LT. 3.2) GO TO 30
   40 Y = 0.7 * X + 1.237
      RETURN
   30 Y = 0.012 * X + 7.2
      RETURN
      END
```

SPECIFICATION STATEMENTS I-6

Specification statements are non-executable; they define the type of a variable or array, specify the amount of storage allocated to each variable according to its type, specify the dimensions of arrays, define methods of sharing storage, and assign initial values to variables.

IMPLICIT	The IMPLICIT statement must precede other specification statements
Type DIMENSION COMMON EQUIVALENCE EXTERNAL LEVEL	If any of these statements appear after the first executable statement or statement function definition, it is ignored and a fatal diagnostic is printed.
DATA	The DATA statement should precede the first executable statement. If a DATA statement appears after the first executable statement, it must not define variables or arrays referenced in the preceding executable statements.

TYPE STATEMENTS

A type statement explicitly defines a variable, array, or function to be integer, real, complex, double precision, or logical. The type statement may be used to supply dimension information. The word TYPE as a prefix is optional.

A symbolic name not explicitly defined in a type, FUNCTION or IMPLICIT statement is implicitly defined as type integer if the first letter of the name is I,J,K,L,M,N; if it is any other letter, the type is real. An explicit definition can override or confirm an implicit definition.

Basic external and intrinsic functions are implicitly typed, and need not appear in a type statement in the user's program. The type of each library function is listed in section 8.

EXPLICIT DECLARATIONS

INTEGER

7

INTEGER $\text{name}_1, \text{name}_2, \ldots, \text{name}_n$

The symbolic names listed are declared to be of type integer.

Example:

```
INTEGER SUM, RESULT, ALIST
```

The variables SUM, RESULT and ALIST are all defined as type integer.

REAL

7

REAL $\text{name}_1, \text{name}_2, \ldots, \text{name}_n$

Example:

```
REAL LIST,JOB3,MASTER4
```

The variables LIST, JOB3, and MASTER4 are all defined as type real.

A real variable is stored in floating point format in one word in memory.

COMPLEX

7

COMPLEX $\text{name}_1, \text{name}_2, \ldots, \text{name}_n$

The symbolic names listed are defined as type complex.

Example:

```
COMPLEX ALPHA, NAM, MASTER, BETA
```

The variables ALPHA, NAM, MASTER, BETA are defined as type complex.

A complex variable is stored as two floating point numbers in two consecutive 60-bit words in memory; the first word is the real part, and the second word is the imaginary part.

DOUBLE PRECISION

7

DOUBLE PRECISION $name_1, name_2, \ldots, name_n$

Double precision variables occupy two consecutive words of memory; the first for the most significant part and the second for the least significant part.

The symbolic names listed are declared to be of type double precision. DOUBLE may be used instead of DOUBLE PRECISION.

Example:

```
DOUBLE PRECISION ALIST, JUNR, BOX4
```

The variables ALIST, JUNR, BOX4 are defined as type double precision.

LOGICAL

7

LOGICAL $name_1, name_2, \ldots, name_n$

The symbolic names listed are defined as type logical.

Example:

```
LOGICAL P,Q,NUMBR4
```

The variables P,Q and NUMBR4 are defined as type logical.

IMPLICIT STATEMENT

7

IMPLICIT $type_1 (ac_1, \ldots, ac_n), \ldots, type_n (ac_1, \ldots, ac_n)$

type	LOGICAL, INTEGER, REAL, DOUBLE PRECISION, or COMPLEX
(ac)	Single alphabetic character, or range of characters represented by the first and last character separated by a minus sign. ac must be enclosed in parentheses.

Example:

```
IMPLICIT REAL (I-M, X), COMPLEX (A-D,N)
```

This statement specifies the type of variables or array elements beginning with the letters ac. Only one IMPLICIT statement may appear in a program unit, and it must precede other specification statements. An IMPLICIT statement in a FUNCTION or SUBROUTINE subprogram affects the type of dummy arguments and the function name, as well as other variables in the subprogram.

Explicit typing of a variable name or array element in a type statement or FUNCTION statement overrides an IMPLICIT specification.

Examples:

```
IMPLICIT INTEGER(A-D,N,R)
DIMENSION GRAD (10,2)
ASUM = BOR + ROR * ANEXT
DECK = CROWN + B
```

The variables ASUM, BOR, ROR, ANEXT, DECK, CROWN and B are of type integer.

In the following example the statement INTEGER A,B,C,D can be replaced by an IMPLICIT statement

```
PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
COMMON A(10),B,C,D
INTEGER A,B,C,D

PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
IMPLICIT INTEGER (A-D)
COMMON A(10),B,C,D
```

Example:

The statement INTEGER A,B,C,D,E(3,4),F,H is replaced by an IMPLICIT statement. A DIMENSION statement is added since an IMPLICIT statement cannot be used to dimension an array. The IMPLICIT statement must also precede all other specification statements.

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      COMMON A(1),B,C,D,  F,G,H
      INTEGER A,B,C,D,E(3,4),F,  H
      EQUIVALENCE (A,E,I)
      NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,VLIST)
      STOP
      END
```

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      IMPLICIT INTEGER (A-F,H)
      DIMENSION E(3,4)
      COMMON A(1),B,C,D,  F,G,H
      EQUIVALENCE (A,E,I)
      NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,VLIST)
      STOP
      END
```

STORAGE ALLOCATION

SUBSCRIPTS

A subscripted symbolic name in the type specification is the name of an array, and the product of the subscripts is the number of elements in the array.

Example:

```
INTEGER ZERO(3,3)
```

defines ZERO as an array of type integer containing 9 integer elements.

```
REAL NEXT(7),ITEM
```

defines NEXT as an array with 7 real elements, and ITEM as a real variable

```
INTEGER CANS(10),NRUMS(7,3),BOX
```

defines CANS as an integer array with 10 elements, NRUMS as an integer array with 21 elements, and BOX as an integer variable

Dimension information should be specified only once for any array name, a second specification is ignored but a warning message is printed.

Examples:

`INTEGER ZERO(3,3)` `DIMENSION ZERO(4,3)`	invalid if both statements appear in the same program; second definition is ignored
`INTEGER CAT` `DIMENSION CAT(4,3,2)`	valid; CAT is an integer array

These statements could be shortened to one statement:

```
INTEGER CAT (4,3,2)
```

DIMENSION STATEMENT

```
      7
      DIMENSION name1(d1),...,namen(dn)
```

d_i	Array declarator, 1-3 integer constants. In a subprogram DIMENSION statement, they can be integer variables.
$name_1,...,name_n$	Symbolic name of an array

```
      PROGRAM SUM (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      DIMENSION INK (10)
      READ (5,100) INK
  100 FORMAT (10I4)
      DO 4 I = 1,10
    4 ITOT = ITOT + INK(I)
      WRITE (6,200) ITOT
  200 FORMAT (10X,*TOTAL = *, I4)
      END
```

DIMENSION is a non-executable statement which defines symbolic names as array names and specifies the bounds of the array.

Example:

```
DIMENSION TOTAL (7,2)
```

TOTAL is defined as a real array of 14 elements.

More than one array can be declared in a single DIMENSION statement.

Example:

```
DIMENSION A(10),B(7,5),C(20,2,4)
```

The number of computer words reserved for an array is determined by the product of the subscripts and the type of the array. For real, integer and logical arrays, the number of words in an array equals the number of elements in the array. For complex and double precision arrays, the number of words reserved is twice the product of the subscripts.

Example:

```
COMPLEX BETA
DIMENSION BETA (2,3)
```

BETA is an array containing six elements; however, BETA has been defined as COMPLEX and two words are used to contain each complex element; therefore, 12 computer words are reserved.

```
REAL NIL
DIMENSION NIL (6,2,2)          reserves 24 words for the array NIL
```

Example:

```
DIMENSION ASUM(10,2)
.
.
.
DIMENSION ASUM (3), VECTOR (7,7)
```

The second specification of ASUM is ignored, and an informative message is printed. The specification for VECTOR is valid and is processed.

ADJUSTABLE DIMENSIONS

Within a subprogram, array dimension specifications may use integer variables, as well as integer constants, provided the array name and all the variable names used for array dimension specifications are dummy arguments of the subprogram. The actual array name and values for the dummy variables are defined by the calling program.

```
  FUNCTION DTOTAL (ARRAY,N)
  DIMENSION ARRAY (N,N)
  DTOTAL = 0.
  DO 1 I = 1,N
1 DTOTAL = DTOTAL + ARRAY (I,I)
  RETURN
  END
```

The above function totals the elements on the major diagonal of any square array. The array name and dimensions are arguments.

A further explanation of adjustable dimensions appears in section 7.

COMMON

```
COMMON/ /v1,...,vn
```

```
COMMON/blkname1/v1,...,vn.../blkname n/v1,...,vn
```

```
      7
      COMMON v1,...,vn
```

blkname	Block name or number enclosed in slashes. A block name is a symbolic name. A block number is 1-7 digits; it must not contain any alphabetic characters. Leading zeros are ignored. 0 is a valid block number. The same block name or number can appear more than once in a COMMON statement or a program unit; the SCOPE loader links all variables in blocks having the same name or number into a single labeled common block.
$v_1,...,v_n$	Variables or array names which can be followed by constant subscripts that declare the dimensions. The variable or array names are assigned to blkname. The COMMON statement can contain one or more block specifications.
//	Denotes a blank common block. If blank common is the first block in the statement, slashes can be omitted.

Example:

```
      PROGRAM CMN (INPUT,OUTPUT)
      COMMON NED (10)
      READ 3,NED
    3 FORMAT (10I3)
      CALL JAVG
      STOP
      END
```

Variables or arrays in a calling program or a subprogram can share the same storage locations with variables or arrays in other subprograms by means of the COMMON statement. Variables and array names are stored in the order in which they appear in the block specification.

COMMON is a non-executable statement. If DIMENSION, COMMON and type specifications appear together, the order is immaterial. The COMMON specification provides up to 125 storage blocks that can be referenced by more than one subprogram. A block of common storage can be labeled by a name or a number. A COMMON statement without a name or number refers to a blank common block. Variables and array elements can appear in both COMMON and EQUIVALENCE statements. A common block of storage can be extended by an EQUIVALENCE statement.

All members of a common block must be allocated to the same level of storage; a fatal diagnostic is issued if conflicting levels are declared. An informative diagnostic is issued if some, but not all, members of a common block are declared in LEVEL statements, and all members are assigned to the declared level.

If any common block member is extended core storage (ECS) resident (section 6, part 3) all members of the block must be ECS resident. No ECS resident elements can appear in blank common.

Block names can be used elsewhere in the program as symbolic names, and they can be used as subprogram names. Numbered common is treated as labeled common. Data stored in common blocks by the DATA statement is available to any subprogram using these blocks.

The length of a common block, other than blank common, must not be increased by a subprogram using the block unless the subprogram is loaded first by the SCOPE loader.

Example:

```
COMMON/BLACK/A(3)
DATA A/1.,2.,3./

COMMON/100/I(4)
DATA I/4,5,6,7/
```

Data may not be entered into blank common blocks by the DATA declaration.

The COMMON statement may contain one or more block specifications:

```
COMMON/X/RAG,TAG/APPA/Y,Z,B(5)
```

RAG and TAG are placed in block X. The array B and Y,Z are placed in block APPA.

Any number of blank common specifications can appear in a program. Blank, named and numbered common blocks are cumulative throughout a program, as illustrated by the following example:

```
COMMON A,B,C/X/Y,Z,D//W,R
.
.
.
COMMON M,N/CAT/ALPHA,BINGO//ADD
```

Have the same effect as the single statement:

```
COMMON A,B,C,W,R,M,N,ADD/X/Y,Z,D/CAT/ALPHA,BINGO
```

Within subprograms, dummy arguments are not allowed in the COMMON statement.

If dimension information for an array is not given in the COMMON statement, it must be declared in a type or DIMENSION statement in that program unit.

Examples:

```
COMMON/DEE/Z(10,4)
```

Specifies the dimensions of the array Z and enters Z into labeled common block DEE.

```
COMMON/BLOKE/ANARAY,B,D
DIMENSION ANARAY(10,2)
```

```
COMMON/Z/X,Y,A
REAL X(7)
```

```
COMMON/HAT/M,N,J(3,4)
DIMENSION J(2,7)
```

In the last example, J is defined as an array (3,4) in the COMMON statement. (2,7) in the DIMENSION statement is ignored and an error message is printed.

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

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

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

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

Example:

```
COMMON/SUM/A,B,C,D  main program

COMMON/SUM/E(3),D   subprogram
```

If the subprogram does not use variables A,B, and C, array E is necessary to space over the area reserved by A,B, and C.

Alternatively, correspondence can be ensured by placing unused variables at the end of the common list.

```
COMMON/SUM/D,A,B,C  main program

COMMON/SUM/D        subprogram
```

If program units share the same common block, they may assign different names and types to the members of the block; but the block name or numbers must remain the same.

Example:

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

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

The subprogram may use different names as in:

```
SUBROUTINE ONE
COMPLEX A
COMMON/TEST/A(10),G(10),K(10)
```

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

EQUIVALENCE STATEMENT

7

EQUIVALENCE $(v_1,\ldots,v_n),\ldots,(v_1,\ldots,v_n)$

$v_1,\ldots,v_n$ are variables, array elements, or array names which can be of different types.

Subscripts must be integer constants. The parentheses are part of the EQUIVALENCE group and must be present. Two or more variables, array elements, or array names can be included in an equivalence group. Dummy arguments and constants are not allowed. More than one equivalence group can appear. ECS resident variables or array elements are not allowed in an equivalence group. Equivalenced variables must be assigned to the same level of storage.

Example:

```
PROGRAM EQUIV (OUTPUT,TAPE6=OUTPUT)
EQUIVALENCE (X,Y),(Z,I)
NAMELIST/OUTPUT/X,Y,Z,I
X=1.
Y=2.
Z=3.
I=4
WRITE(6,OUTPUT)
STOP
END
```

```
$OUTPUT

X        =  0.2E+01,

Y        =  0.2E+01,

Z        =  0.0,

I        =  4,

$END
```

An explanation of this example appears in part 2.

EQUIVALENCE is a non-executable statement and must appear before all executable statements in a program unit. If it appears after the first executable statement, a fatal diagnostic is printed. Variables or array elements not mentioned in an EQUIVALENCE statement are assigned unique locations.

EQUIVALENCE assigns two or more variables in the same program unit to the same storage location (as opposed to COMMON which assigns two variables in different program units to the same location).

Example:

```
DIMENSION JAN(6),BILL(10)
EQUIVALENCE (IRON,MAT,ZERO), (JAN(5),BILL(2)),(A,B,C)
```

The variables IRON, MAT and ZERO share the same location, the fifth element in array JAN and the second element in array BILL share the same location, and the variables A,B and C share the same location.

When an element of an array is referred to in an EQUIVALENCE statement, the relative locations of the other array elements are, thereby, defined also.

Example:

```
DIMENSION Y(4), B(3,2)
EQUIVALENCE (Y,B(1,2)), (X,Y(4))
```

This EQUIVALENCE statement causes storage to be shared by the first element in Y and the fourth element in B and, similarly, the variable X and the fourth element in Y. Storage will be as follows:

B(1,1)		
B(2,1)		
B(3,1)		
B(1,2)	Y(1)	
B(2,2)	Y(2)	
B(3,2)	Y(3)	
	Y(4)	X

The statement EQUIVALENCE(A,B),(B,C) means the same as EQUIVALENCE (A,B,C).

When no array subscript is given, it is assumed to be 1.

```
DIMENSION ZEBRA(10)
EQUIVALENCE (ZEBRA,TIGER)
```

Means the same as the statements:

```
DIMENSION ZEBRA(10)
EQUIVALENCE (ZEBRA(1),TIGER)
```

A logical, integer, or real entity equivalenced to a double precision or complex entity shares the same location as the real or most significant part of the complex or double precision entity.

An array with multiple dimensions may be referenced with a single subscript. The location of the element in the array may be determined by the following method:

```
DIMENSION A(K,M,N)
```

The position of element A(k,m,n) is given by:

```
A+(k-1+K*(m-1+M*(n-1))*E
```

E is 1 if A is real, integer or logical; E is 2 if A is complex or double precision.

Example:

```
DIMENSION AVERAG(2,3,4),TERM(7)
EQUIVALENCE (AVERAG(8),TERM(2))
```

Elements AVERAG (2,1,2) and TERM(2) share the same locations.

Two or more arrays can share the same storage locations.

Example:

```
    DIMENSION ITIN(10,10),TAX(100)
    EQUIVALENCE(ITIN,TAX)
    .
    .
    .
500 READ (5,40)ITIN
    .
    .
    .
600 READ (5,70) TAX
```

The EQUIVALENCE declaration assigns the first elements of arrays TIN and TAX to the same location. READ statement 500 stores the array TIN in consecutive locations. Before READ statement 600 is executed, all operations involving ITIN should be completed; as the values of array TAX are read into the storage locations previously occupied by ITIN.

Lengths of arrays need not be equal.

Examples:

```
DIMENSION ZERO1(10,5),ZERO2(3,3)
EQUIVALENCE (ZERO1,ZERO2)
```
is a legal EQUIVALENCE statement

```
EQUIVALENCE (ITEM,TEMP)
```

The integer variable ITEM and the real variable TEMP share the same location; therefore, the same location may be referred to as either integer or real. However, the integer and real internal formats differ; therefore the values will not be the same.

Example:

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      COMMON A(1),B,C,D,  F,G,H
      INTEGER A,B,C,D,E(3,4),F,  H
      EQUIVALENCE (A,E,I)
      NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,VLIST)
      STOP
      END
```

Output from Program COME:

```
$VLIST

A        =  1,

B        =  2,

C        =  3,

D        =  4,

E        =  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11,  12,

F        =  5,

G        =  0.0,

H        =  7,

I        =  1,

$END
```

An explanation of this example appears in part 2.

EQUIVALENCE AND COMMON

Variables, array elements, and arrays may appear in both COMMON and EQUIVALENCE statements. A common block of storage may be extended by an EQUIVALENCE statement.

Example:

```
COMMON/HAT/A(4),C
DIMENSION B(5)
EQUIVALENCE (A(2),B(1))
```

Common block HAT will extend from A(1) to B(5):

/HAT/

Origin	A(1)	
	A(2)	B(1)
	A(3)	B(2)
	A(4)	B(3)
	C	B(4)
		B(5)

EQUIVALENCE statements which extend the origin of a common block are not allowed, however.

Example:

```
COMMON/DESK/E,F,G
DIMENSION H(4)
EQUIVALENCE (E,H(3))
```

The above EQUIVALENCE statement is illegal because H(1) and H(2) extend the start of the common block DESK:

/DESK/

		H(1)
		H(2)
origin	E	H(3)
	F	H(4)
	G	

No two elements in the same common block or in different common blocks may be set equivalent to each other.

Examples:

```
COMMON A,B,C
EQUIVALENCE (A,B)            illegal

COMMON /HAT/ A(4),C /X/ Y,Z
EQUIVALENCE (C,Y)            illegal
```

LEVEL STATEMENT

7

LEVEL n, $a_1, \ldots, a_n$

$a_1,\ldots,a_n$	List of variables or array names separated by commas
n	Unsigned integer 1, 2, or 3 indicating level to which list is to be allocated.
1	Small core memory resident (SCM)
2	Large core memory resident (LCM). Directly addressable (or word addressable)
3	Large core memory resident, accessed by block transfer to or from small core memory through MOVLEV subroutine call
1	Central memory resident
2	Central memory resident
3	Extended core storage resident, accessed by block transfer to or from central memory through MOVLEV subroutine call

This statement assigns variables or array names to the level n. LEVEL statements must precede the first executable statement in a program unit. Names of variables and arrays which do not appear in a LEVEL statement are allocated to small core memory (level 1) in 7600, and central memory (levels 1 and 2) in 6000 series computers.

No dimension or type information may be included in the LEVEL statement.

Variables and arrays appearing in a LEVEL statement can appear in DATA, DIMENSION, EQUIVALENCE, COMMON, type, SUBROUTINE and FUNCTION statements. Data assigned to levels 2 and 3 must appear also in COMMON statements or as dummy arguments in SUBROUTINE statements.

Data assigned to level 3 can be referenced only in COMMON, CALL, SUBROUTINE, FUNCTION and DIMENSION statements. Level 3 items cannot be used in expressions.

No restrictions are imposed on the way in which reference is made to variables or arrays allocated to levels 1 and 2.

If the level of any variable is multiply defined, the level first declared is assumed; and a warning diagnostic is printed.

All members of a common block must be assigned to the same level; a fatal diagnostic is issued if conflicting levels are declared. If some, but not all, members of a common block are declared in a LEVEL statement, all are assigned to the declared level, and an informative diagnostic is printed.

If a variable or array name declared in a LEVEL statement appears as an actual argument in a CALL statement, the corresponding dummy argument must be allocated to the same level in the called subprogram.

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

Example:

```
DIMENSION E(500),B(500),CM(1000)
LEVEL 3, E,B
COMMON /ECSBLK/ E,B
.
.
.
CALL MOVLEV (CM,E,1000)
```

The LEVEL statement allocates arrays E and B to extended core storage or to LCM. They are assigned to a named common block, ECSBLK. Starting at location CM (the first word address of the array CM), 1000 words of central memory are transferred to the two arrays E and B in extended core storage or LCM by the library routine MOVLEV.

EXTERNAL STATEMENT

7

EXTERNAL $name_1, \ldots, name_n$

$name_1, \ldots, name_n$ — **Subprogram names**

Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program.

Any name used as an actual argument in a call is assumed to be a variable or array unless it appears in an EXTERNAL statement. An EXTERNAL statement must be used even if the subprogram concerned is a standard system function, such as SQRT. However, an EXTERNAL statement is not required for intrinsic functions used as actual arguments. If an intrinsic function name appears in an EXTERNAL statement, the user must supply the function.

Example:

Calling Program	**Subprogram**
`EXTERNAL SIN, SQRT`	`SUBROUTINE SUBRT (A,B,C)`
`CALL SUBRT(2.0,SIN,RESULT)`	`X=A+3.14159/2.`
`WRITE (6,100) RESULT`	`C=B(X)`
`100 FORMAT (F7.3)`	`RETURN`
`CALL SUBRT(2.0,SQRT,RESULT)`	`END`
`WRITE (6,100)RESULT`	
`STOP`	
`END`	

First the sine, then the square root are computed; and in each case, the value is returned in RESULT. The EXTERNAL statement must precede the first executable statement, and always appears in the calling program. (It may not be used with statement functions.)

A function call that provides values for an actual argument does not need an EXTERNAL statement.

Example:

Calling Program

```
CALL SUBRT(SIN(X),RESULT)
```

Subprogram

```
SUBROUTINE SUBRT(A,B)
.
.
.
B=A
.
.
.
END
```

An EXTERNAL statement is not required because the function SIN is not the argument of the subprogram; the evaluated result of SIN(X) becomes the argument.

Example:

```
      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      NAMELIST/V/X,Y,AA,AM
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
      AA=PVAL(12,AVG)
      AM=PVAL(12,MULT)
      WRITE(6,V)
      STOP
      END
```

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
      COMMON A(100)
      AVG=0.
      DO 1 I = 1,J
    1 AVG=AVG+A(I)
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
      REAL FUNCTION MULT(J)
      COMMON ARRAY(12)
      MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
      RETURN
      E  N  D
```

An explanation of this example appears in part 2.

DATA STATEMENT

DATA (var = dlist),. . . , (var = dlist)

7

DATA $vlist_1/dlist_1/,\ldots, vlist_n/dlist_n/$

var — Variable, array element, array name or implied DO

vlist — List of array names, array elements, variable names, or an implied DO loop, separated by commas. Array elements must have integer constant subscripts, unless they appear in an implied DO loop.

dlist — One or more of the following forms separated by commas:

> constant
> (constant list)
> rf*constant
> rf*(constant list)
> rf(constant list)
>
> constant list — List of constants separated by commas
>
> rf — Integer constant. The constant or constant list is repeated the number of times indicated by rf.

The DATA statement assigns to variables or array elements initial values which are compiled into the object program from source program statements. When source program execution begins, these values are assumed by the variables or arrays. Any variables not assigned values by the DATA statement are unspecified.

Example:

```
DATA A,B,C/3.,27.5,5.0/
```
assigns 3. to A, 27.5 to B, 5.0 to C

The DATA statement is non-executable and should, as good programming practice, precede the first executable statement in the program or subprogram. The DATA statement must follow all specification statements. One DATA statement must not contain both forms of the list (vlist/dlist/ and var=dlist).

Dummy arguments or elements in blank common cannot be assigned values in a DATA declaration.

In the DATA statement, the type of constant stored is determined by the structure of the constant rather than by the type of the variable in the statement.

Example:

```
DATA IRUN/10./
```

10. is stored as a real constant, not as an integer, as might be expected from the form of the symbolic name IRUN.

```
DATA ITEM, JOB/10,10./
```

An integer constant 10 is stored in ITEM, and a real constant 10. is stored in JOB. The two constants will be stored differently:

0000000000000000012 integer

1723500000000000000 real

Any future use of the integer variable JOB could produce erroneous results.

The value of the item in the data list is assigned to the corresponding variable in the variable list. The number of items in the data list should agree with the number of variables in the variable list.

Example:

```
DATA A,B,C/7.,8.,9./
```

7. 8. and 9. are assigned to A, B, and C respectively.

If the data list contains more items than the variable list, excess items are ignored, and an informative diagnostic is printed.

Example:

```
COMMON/LABEL/A(3)
DATA A/1.,2.,3.,4./
```

Constants 1.,2. and 3. are stored in array locations A, A+1, A+2; constant 4. is discarded; and an informative message is printed.

If the data list contains fewer items than the variable list, the value of the remaining variable is not defined, and an informative diagnostic is printed.

Example:

```
COMMON/NAME/C(3)
DATA C/1.,2./
```

Constants 1. and 2. are stored in locations C(1) and C(2); the content of C(3), that is, location C+2 is not defined.

The implied DO loop may be used to store values into arrays.

Example:

```
REAL ANARAY(10)
DATA (ANARAY(I),I = 1,10)/1.,2.,3.,7*2.5/
```

Values stored in array ANARAY:

ANARAY(1)	1.
	2.
	3.
	2.5
	2.5
	2.5
	2.5
	2.5
	2.5
ANARAY(10)	2.5

When an implied DO is used to store values into arrays, only one array name can be used within the implied DO nest. The array name in the implied DO nest is not related in any way to an array of the same name in the same program unit.

Example:

Invalid: `DATA (A(I),B(I),I=1,3)/1.,2.,3.,4.,5.,6./`

Valid: `DATA ((C(I,J),J=1,4,3),I=1,3)/1.,2.,3.,4.,5.,6./`

Example:

```
DATA A,B,C,D/4*2.7/
```

The value 2.7 is assigned to the variable A,B,C and D. If the number preceding the asterisk is not an integer, a fatal diagnostic is printed.

The following examples illustrate the use of the DATA statement:

```
COMPLEX PROTER (4)
DATA PROTER/4*((1.0,2.0))/
```

4 complex constants (1.0,2.0) are stored in the ARRAY PROTER

1.0
2.0

1.0
2.0

1.0
2.0

1.0
2.0

Note: (1.0,2.0) is a complex constant, 2*(1.0,2.0) means repeat a constant list containing elements 1.0 and 2.0 twice, 2*((1.0,2.0)) means repeat the complex constant (1.0,2.0) twice.

Example:

```
DATA A(1,3)/16.239/
```

16.239 is stored in the element in the first row, 3rd column of array A.

```
DIMENSION B(10)
DATA B/000077B,000064B,3*000005B,5*000200B/
```

The following octal constants are stored in ARRAY B:

```
 77B
 64B
  5B
  5B
  5B
200B
200B
200B
200B
200B
```

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

ARRAY C contains the following elements:

```
 3.6
10.5
10.5
10.5
```

```
LOGICAL L(4)
DATA L/4*.TRUE./
```

The logical variables in array L are set to the value .TRUE.

Examples of alternative form of DATA statement:

```
DATA (X=3),(Y=5)

INTEGER ARAY(5)
DATA (A=7),(B=200.),(ARAY=1,2,7,50,3)

COMMON/BOX/ARAY4(3,4,5)
DATA (ARAY4(1,3,5)=22.5)

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

initializes:

D3(1) = 5.
D3(2) = 6.
D3(3) = 7.
D3(4) = 8.

and sets the entire POQ array to zero.

When constants in a data list are enclosed in parentheses and preceded by an integer constant, the list is repeated the number of times indicated by the integer constant. If the repeat constant is not an integer, a compiler error message is printed.

Example:

```
DIMENSION B(10)
DATA((B(I),I=1,10)=15.,2.,3.7,7(4.32))

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
```

Data may not be entered into blank common with a DATA statement.

When a Hollerith specification is used in a DATA statement, it should not exceed 10 characters.

For example, to store the following values in an array A

A(1) = 1234567890

A(2) = ABCDEFGHIJ

A(3) = KLMNOPQRST

A(4) = UVWXYZ+- *

The following statements should be used:

```
DIMENSION A(4)
DATA A/10H1234567890,10HABCDEFGHIJ,10HKLMNOPQRST,10HUVWXYZ+- */
```

The following statements would not produce the desired result:

```
DIMENSION A(4)
DATA A/20H1234567890ABCDEFGHIJ,20HKLMNOPQRSTUVWXYZ+- */
```

They would initialize

A(1) 1234567890

A(2) KLMNOPQRST

A(3) UVWXYZ+- *

A(4) undefined

BLOCK DATA SUBPROGRAM

Data may be entered into labeled or numbered common (but not blank common) prior to program execution by the use of the BLOCK DATA subprogram. This subprogram should contain only the DATA, COMMON, DIMENSION, EQUIVALENCE, type, and END statements associated with the data defined. Any executable statements will be ignored, and a warning printed.

A BLOCK DATA subprogram has one of the following formats:

```
BLOCK DATA name
.
.
.
END

BLOCK DATA
.
.
.
END
```

name is any legal FORTRAN name. It identifies the BLOCK DATA subprogram if more than one BLOCK DATA subprogram is compiled. If the user does not name the block, it is given the name BLKDATA.

DATA may be entered into more than one block of common in one subprogram.

Example:

```
BLOCK DATA ANAME
COMMON/CAT/X,Y,Z/DEF/R,S,T
COMPLEX X,Y
DATA X,Y/2*((1.0,2.7))/,R/7.6543/
END
```

Z is in block CAT, and S and T are in DEF; although no initial data values are defined for them.

The DATA statement must follow the specification statements.

```
 BLOCK DATA
 COMMON/ABC/A(5),B,C/BILL/D,E,F
 COMPLEX D,E
 DOUBLE PRECISION F
 DATA (A(L),L=1,5)/2.3,3.4,3*7.1/,B/2034.756/,D,E,F/2*((1.0,2.5)),
S 7.86972415872D30/
 END
```

MAIN PROGRAM AND SUBPROGRAMS

A FORTRAN program may be written with or without subprograms. One main program is required in any executable FORTRAN program; any number of subprograms may be included.

MAIN PROGRAM

A main program should begin with the PROGRAM statement. If this statement is omitted from the main program, the program is assumed to have the name START., and files INPUT and OUTPUT are assumed.

PROGRAM STATEMENT

Under the SCOPE operating system, all data used by a program must have a file name. The FORTRAN programmer should list this file name in the PROGRAM statement. The FORTRAN compiler adds the characters TAPE as a prefix to each logical unit number referenced in the user's program to form a file name. For example, logical unit 3 is assigned the file name TAPE3, and the programmer should list the file name TAPE3 in the PROGRAM statement if he references logical unit 3 in his program. SCOPE file names INPUT, OUTPUT and PUNCH should appear in the PROGRAM statement when READ, WRITE and PUNCH statements are used in a program.

The file name must appear in the PROGRAM statement of the main program even if the read or write statement is in a subprogram.

```
      7
      PROGRAM name (file,...,file)
```

name	Must be a unique symbolic name within the main program and cannot be used as a subprogram name. It will be the entry point name and the object deck name for the SCOPE loader.
(file,...,file)	Names of all input/output files required by the main program and its subprograms; maximum number of file names is 50. All internal file names used in input/output statements should be declared. If the program is to be loaded as an overlay (but not as the main overlay) this parenthetical list must be omitted.
file	1-6 character file name

file=n	n is a decimal number specifying the buffer length. It must appear with the first reference to the file in the PROGRAM statement. If no buffer length is specified, a default double buffer (2002B) is assumed. A buffer length of zero can be specified. For example, PROGRAM X (TAPE=0) is a legal statement. If file=n is specified in a 7600 program, it is ignored.
$file_n = file_m$	Files will be made equivalent. File m must have been previously defined. File names may be made equivalent at compile time, but file m must have been previously defined in the same PROGRAM statement. All references in the source code to file n refer to file m. Since m and n refer to the same file, any buffer length specified applies to both file names.

Example:

```
PROGRAM ORB (INPUT,OUTPUT=1000,TAPE1=INPUT,TAPE2=OUTPUT)
```

All input/output statements which reference TAPE1 will instead reference INPUT, and all listable output normally recorded on TAPE2 would be transmitted to the file named OUTPUT.

Only one level of parentheses is allowed in the PROGRAM statement. The PROGRAM statement is scanned from left to right.

Example:

```
PROGRAM SORT (INPUT,OUTPUT,TAPE5=INPUT,COMPILE=4000,TAPE20=COMPILE)
```

At compile time, the file names should satisfy the following conditions (file names can be changed at execution time by SCOPE control cards). If these conditions are not met, a warning diagnostic is printed:

1. File name INPUT should be defined if any READ fn, iolist statement is included in the program.

2. File name OUTPUT should be defined if any print statement is included. If execution error messages are to be listed, OUTPUT must be included.

3. File name PUNCH should be defined if any PUNCH statement is included in the program.

4. File name TAPEu (u is an integer constant 1-99) should be defined if any input/output statement involving unit u appears in the program. At execution time, if u is a variable, there must be a file name TAPEu for each value u may assume.

At execution time, if file names have not been defined in the PROGRAM statement, they must be defined by SCOPE control cards (refer to File Name Handling by System, section 3, part 3). If they are not defined, a fatal error results and the message UNDEFINED FILE NAME is printed.

The characters TAPE are added as a prefix to each logical unit number in the user's program. Logical unit 3 is assigned the file name TAPE3, logical unit 4 is assigned the file name TAPE4. Note, TAPE5 and TAPE05 are not the same file name.

A logical unit number is assigned by writing TAPEu = filenam, where filenam is the name of the file with which the logical unit number is to be associated.

Examples:

```
      PROGRAM X (INPUT,TAPE5=INPUT)

      PROGRAM Y (OUTPUT,TAPE2=OUTPUT)

      PROGRAM OUT(OUTPUT,TAPE6=OUTPUT)
      .
      .
      .
      WRITE(6,200)A,B,C
  200 FORMAT (1H1,3F7.3)
```

Logical unit 6 must be declared as TAPE6 in the PROGRAM statement.

```
      PROGRAM IN(INPUT,TAPE5=INPUT)
      .
      .
      .
      READ(5,100)A,B,C
  100 FORMAT (3F7.3)
```

This statement reads from logical unit 5, it is declared in the PROGRAM statement as TAPE5.

When a file name is made equivalent to another file, the file name appearing to the right of an equals sign must have been previously declared in the same statement.

Example:

In the following statement, INPUT and OUTPUT are defined before they appear to the right of the equals sign. TAPE5 becomes an alternate name for the file INPUT, and TAPE6 becomes an alternate name for OUTPUT.

```
PROGRAM SAMPLE (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
```

Example:

```
PROGRAM JIM(INPUT,TAPE19=INPUT)
```

TAPE19=INPUT must be preceded in the same statement by INPUT (or INPUT = buffer length)

If any of the following statements are used in a program or its subprograms, the logical unit number, u, must appear as file name TAPEu in the program statement:

```
WRITE (u) iolist            ENDFILE u
WRITE (u,fn) iolist         BACKSPACE u
READ (u) iolist             REWIND u
READ (u,fn) iolist          BUFFER IN (u,p) (a,b)
                            BUFFER OUT (u,p) (a,b)
```

If u is a variable, there must be a file name TAPEu for each value u can assume in the source program.

Example:

```
    PROGRAM KAY(INPUT,OUTPUT,TAPE60=INPUT,TAPE61=OUTPUT)
    .
    .
    .
    READ(60,100)ALIST
100 FORMAT (F7.3)
    .
    .
    .
    WRITE (61,200)ALIST
200 FORMAT (1H0,F7.3)
```

Example:

```
    PROGRAM JIM(TAPE1,TAPE2,TAPE3,TAPE5)
    .
    .
    .
    N=2
    .
    .
    .
    READ(N)
    .
    .
    .
    N=2+1
    .
    .
    .
    READ(N)
```

SUBPROGRAMS

A subprogram is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is a specification subprogram as described in Section 6. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram.

Procedure subprograms are of two types: subroutine and function. Function subprograms return a single value to the expression containing the function's name. The four kinds of functions are:

Statement functions FUNCTION subprograms	user defined
Intrinsic functions (in-line functions) library functions	system supplied

Subroutine subprograms may return a number of values (or none at all); they are referenced by a CALL statement. The two kinds of subroutines are:

User subroutine

Library subroutine

Subprograms are defined separately from the calling program and may be compiled independently of the main program. They are complete program units conforming to all the rules of FORTRAN programs. The term program unit refers to either a main program or a subprogram.

A subprogram may call other subprograms as long as it does not directly or indirectly call itself. For example, if program A calls program B, B may not call A. A calling program is a program unit which calls a subprogram.

Subprogram definition statements declare certain names to be dummies representing the arguments of the subprogram—these are called dummy arguments. They are used as ordinary names within the defining subprogram and indicate the number, type and order of the arguments and how they are used. The dummy arguments are replaced by the actual arguments when the subprogram is executed. Dummy arguments may not appear in COMMON, EQUIVALENCE, or DATA statements.

Actual parameters appear in subroutine calls

```
CALL SUB3 (7.,CAT, 8.932)
```

or function references

```
A = B + ROOT (6.5,7.,BOX)
```

FUNCTION SUBPROGRAM

DEFINING A FUNCTION SUBPROGRAM

```
      FUNCTION name (p1,...,pn)
```

7

```
      type FUNCTION name (p1,...,pn)
```

$p_1,\ldots,p_n$	Dummy arguments which should agree in order, number, and type with the actual arguments in the calling program. At least one argument is required; a maximum of 63 is allowed.
type	The type may be REAL, INTEGER, DOUBLE PRECISION, (the word PRECISION is optional), COMPLEX or LOGICAL. When type is omitted, and no IMPLICIT statement appears in that program unit, the type of the function result is determined by the first character of the function name.
name	FUNCTION name. It must not appear in any non- executable statement other than the FUNCTION statement in the subprogram.

Dummy arguments may be the names of arrays, variables, and subprograms. Since all names are local to the subprogram containing them, dummy arguments may be the same as names appearing outside the subprogram. A dummy argument must not appear in COMMON, EQUIVALENCE or DATA statements within the function subprogram.

The programmer can define a sequence of statements as a function. A function subprogram begins with a FUNCTION declaration and returns control to the calling program when a RETURN statement in the function subprogram is encountered. Execution of the FUNCTION subprogram results in a single value which is returned to the main program through the function name.

The name of the function must be assigned a value within the function subprogram; if it is not assigned a value, a warning diagnostic is printed. This value is the value of the function.

If an END statement is encountered in the FUNCTION subprogram, a RETURN is assumed.

A function must not, directly or indirectly reference itself.

FUNCTION SUBPROGRAM REFERENCE

A function is referenced when the name of a function appears in an arithmetic, logical or masking expression. A function reference transfers control to the function subprogram, and the values of the actual arguments are substituted for the dummy arguments.

Actual arguments may be arithmetic or logical expressions, constants, variables, array names, array element names, SUBROUTINE subprogram names, an external function name (not an intrinsic function or statement function), or function reference (the function reference is a special case of an arithmetic expression), or a Hollerith constant, or an ECS variable, array or array element name, or an LCM variable, array name or array element name.

Example:

```
    .                                         FUNCTION GRATER(A,B)
    .                                         IF (A.GT.B)1,2
    .                                       1 GRATER=A-B
    W(I,J)=FA+FB-GRATER(C-D,3*AX/BX)          RETURN
    .                                       2 GRATER=A+B
    .                                         RETURN
    .                                         END
```

When a RETURN statement in the function subprogram is executed, and control is returned to the statement containing the function reference, if A is greater than B the value of A-B, in this case, C-D-3*AX/BX is returned to the main program and used in the evaluation of the expression. If A is less than B, the value of A+B (C-D+3*AX/BX) is returned to the main program.

A function reference may appear anywhere in an expression that an operand may be used.

The name of a function must not appear in a DIMENSION declaration. Dummy arguments representing array names must appear within the subprogram in a DIMENSION or type statement giving dimension information. If dummy arguments are not dimensioned, they cannot be referenced as an array in the subprogram.

If the subscripts of an array in the subprogram are to agree with the subscripts in the calling program, the dimensions in the subprogram must be the same as those in the calling routine. If array dimensions between subprogram and calling program differ, the user must be aware of the arrangement of arrays in storage (Common, section 6 and Arrays, section 2).

Example:

```
    .
    .
    .
    DIMENSION ARY (5,5)                       FUNCTION DIAG (A,N)
    .                                         DIMENSION A(5,5)
    .                                         DIAG=A(1,1)
    .                                         DO 70 I=1,N
    RES=DIAG(ARY,5)**2                     70 DIAG=DIAG*A(I,I)
    .                                         RETURN
    .                                         END
    .
```

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

The FUNCTION subprogram can define or redefine one or more of its arguments to return results (as well as the value of the function) to the calling program.

Adjustable dimensions are permitted in FUNCTION subprograms.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function or a subroutine name.

CONFLICTS WITH LIBRARY NAMES

If the user writes a FUNCTION subprogram with the same name as a library or intrinsic function, he should be aware of two function properties:

Library external and intrinsic functions are implicitly typed.

Basic external functions may be called by value.

The FORTRAN library is listed in section 8. FORTRAN Extended provides functions additional to those specified in ANSI.

Library function names are implicitly typed. For example, DSIN and DSQRT are type DOUBLE PRECISION. If the user writes a function with the name DSIN, his function will override the library function DSIN; but it is taken as type DOUBLE PRECISION in the calling program unit unless a different type is specified. The types of all external functions are listed in section 8.

Any user written function which has the same name as a basic external function, and is called by value, must appear in an EXTERNAL statement in the calling program unit. The list of external functions called by value appear in table 8-2 in section 8.

CALL BY NAME AND CALL BY VALUE

To increase speed, arguments to library functions are normally passed to subprograms by placing their values in the registers. This method is call by value. For user defined subprograms, the address of the arguments are passed to the subprogram. This method is call by name. A user supplied external function is always called by name. When the control card options T, D, or OPT=0 are specified on the control card, library subprograms are called by name also.

If the user defines a function with the same name as a basic external function, the name of his function overrides the library function, and the user function is referenced. However, the name still retains the type of the library external function and the call-by-value property. Therefore, unless the user defined function appears in an EXTERNAL statement, results may be undefined.

For example, the situation could arise where a user, having defined a function which has the same name as a library function, debugs his program using the call-by-name option T. During the debug phase no problem would arise. However, after the program is debugged, he might select the call-by- value option for faster object code; at this point, results would be undefined.

If the name of an intrinsic function is used in any context other than a function reference, the user's definition overrides the system definition. However, a user supplied FUNCTION subprogram with the same name as an intrinsic function will be ignored, and the intrinsic function is used if the name does not appear in a type statement which differs from the type of the intrinsic function or in an EXTERNAL statement in the calling program.

STATEMENT FUNCTION

DEFINING A STATEMENT FUNCTION

```
      7
      name (p1,p2,p3,..., pn) = expression
```

name — Type of the function is determined by the type of the function name, unless it appears in a type statement.

$p_1,...,p_n$ — Dummy arguments must be simple variable names. At least one argument is required; a maximum of 63 is allowed. These arguments should agree in order, number, and type with the actual arguments used in the function reference.

expression — Any arithmetic, masking, relational, or logical expression may be used. It may contain references to library functions, statement functions, or function subprograms. Names in the expression which do not represent arguments have the same value as they have outside the function (they are normal variables).

The definition of a statement function is contained in a single statement, and it applies only to the program or subprogram containing the definition. It consists of one statement and produces only one result.

Statement function names must not appear in DIMENSION, EQUIVALENCE, COMMON or EXTERNAL statements; they can appear in a type declaration but cannot be dimensioned. Statement function names must not appear as actual or dummy arguments. If the function name is type logical, the expression must be logical. For other types, if the function name and expression differ, conversion is performed as part of the function.

A statement function must precede the first executable statement and it must follow all specification statements (DIMENSION, type, etc.). A statement function must not reference itself. For example, R(I) = R(I) *R(I-1) is illegal unless R is an array name.

Examples:

```
LOGICAL C,P,EQV
EQV(C,P) = (C.AND.P).OR.(.NOT.C.AND..NOT.P)

COMPLEX Z,F(10,10)
Z(A,I) = (3.2,0.9)*EXP(A)*SIN(A)+(2.0,1.)*EXP(Y)*COS(B)+F(I,J)

GROS(R,HRS,OTHERS) = R*HRS + R* .5*OTHERS
```

STATEMENT FUNCTION REFERENCE

The statement function only defines the function; it does not result in any computation.

The value of the function is computed using the values of the actual arguments. The actual arguments are substituted when a statement function reference is made; they may be any arithmetic expressions. Statement function names should not appear in an EXTERNAL statement.

For example, to compute one root of the quadratic equation $ax^2+bx+c=0$, given values of a, b and c, an arithmetic statement function can be defined as follows:

```
ROOT (A,B,C)=(-B+SQRT(B*B-4.*A*C))/(2.0*A)
```

When the function is used in an expression, actual arguments are substituted for the dummy arguments A,B,C.

```
RESA = ROOT (6.5,7.,1.)
```

is equivalent to writing

```
RESA = (-7.+SQRT(7.*7.-4.0*6.5*1.0))/(2.0*6.5)
```

or

```
TAB = 3.7 * ROOT (CAT, 8.2, TEMP) + BILL
```

Wherever the statement function ROOT (A,B,C) is referenced, the definition of that function—in this case (-B+SQRT(B*B-4.*A*C))/(2.*A)—is evaluated using the current values of the arguments A,B,C.

Examples:

Statement Function Definitions	Statement Function References
`ADD(X,Y,C,D)=X+Y+C+D`	`RES1=GROSS-ADD(TAX,FICA,INS,RES3)`
`AVERGE(O,P,Q,R)=(O+P+Q+R)/4`	`GRADE=AVERGE(TEST1,TEST2,TEST3,` `TEST4)+MID`

`LOGICAL A,B,EQV`
`EQV(A,B)=(A.AND.B).OR.`
`(.NOT.A.AND..NOT.B)`
	`TEST=EQV(MAX,MIN).AND.ZED`

`COMPLEX Z`
`Z(X,Y)=(1.,0.)*EXP(X)*COS(Y)`
`+(0.,1.)*EXP(X)*SIN(Y)`
	`RESULT=(Z(BETZ,GAMMA(I+K))**2-1.)`
	`/SQRT(TWOPIE)`

Here, the statement function is used to substitute a library function name in a program containing an alternate name for this library function.

```
SINF(X)=SIN(X)        statement function definition
     .
     .
     .
A=SINF(3.0+B)+7.
```

The above sequence generates exactly the same object code as:

```
A=SIN(3.0+B)+7.
```

During compilation, the statement function definition is retained by the compiler. Whenever the function is referenced, instructions are generated in line to evaluate the function (as opposed to FUNCTION subprograms for which a branch instruction is generated at each reference). The expansion of a statement function is similar to the expansion of an assembly language macro. Thus the statement function does not reduce execution speed or efficiency.

SUBROUTINE SUBPROGRAMS

DEFINING A SUBROUTINE SUBPROGRAM

```
SUBROUTINE name (p1,p2,...,pn)
```

```
SUBROUTINE name
```

```
SUBROUTINE name (p1,p2,...,pn), RETURNS (b1,b2,...,bm)
```

```
SUBROUTINE name, RETURNS (b1,b2,...,bm)
```

name	Symbolic name of the SUBROUTINE
$p_1,..., p_n$	Dummy arguments which must agree in order, number and type with the actual arguments passed to the subprogram at run time. A maximum of 63 is allowed. The argument list is optional. Dummy arguments can be the names of arrays, simple variables, library functions, or subprograms. Since dummy arguments are local to the subprogram containing them, they may be the same as names appearing outside the subprogram. A dummy argument must not appear in a COMMON, EQUIVALENCE, or DATA statement within the subroutine.
$b_1,...,b_m$	Dummy statement numbers which must agree in order and number with the actual statement numbers passed to the SUBROUTINE at run time. The variables $b_1,...,b_m$ cannot be defined in the SUBROUTINE.

A SUBROUTINE subprogram can be referred to only by a CALL statement. It starts with a SUBROUTINE statement and returns control to the calling program through one or more RETURN statements. The subprogram name is not used to return results to the calling program and does not determine the type of the subprogram. Values are passed by one or more arguments or through common (refer to SUBPROGRAMS and COMMON).

Dummy arguments which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. If an array name without subscripts is used as an actual argument in a CALL statement and the corresponding dummy argument has not been declared an array in the subprogram, the first element of the array is used in the subprogram. Adjustable dimensions are permitted in SUBROUTINE subprograms.

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 PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement.

The SUBROUTINE name must not appear in any other statement in the same subprogram.

Example:

Calling Program

```
   .
   .
   .
   CALL PGM1(A,B,C),
  XRETURNS (5,10)
   .
   .
   .
 5 B=SQRT(A*C)
   .
   .
   .
10 CALL PGM2 (D,E)
   .
   .
   .
```

Subprogram

```
    SUBROUTINE PGM1(X,Y,Z),
   XRETURNS (M,N)
    U=X**Y
    X=Z+X*Y
20  IF (U+X) 25, 30, 35
25  RETURN M          Return is to statement 5 in calling program
30  RETURN N          Return is to statement 10 in calling program
35  Z=Z+(X*Y)
    RETURN            Return is to statement following CALL PGM1
    END
```

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

The RETURN statement is described in section 5.

REFERENCING A SUBROUTINE SUBPROGRAM

CALL STATEMENT

The CALL statement causes a SUBROUTINE subprogram to be executed.

```
CALL name

CALL name (p1,...,pn)

CALL name (p1,...,pn), RETURNS (b1,...,bm)

7
CALL name, RETURNS (b1,...,bm)
```

name — Name of subroutine called must not appear in any specification statement in the calling program except an EXTERNAL statement.

$p_1,...,p_n$ — Actual arguments which must correspond in order, number, and type with those specified in the SUBROUTINE statement.

$b_1,...,b_m$ — Numbers of statements in the calling program or subprogram to which control returns. They correspond in order and number with the dummy statement numbers in the subroutine. If alternate exits are taken from the subroutine, $b_1,...,b_m$ must be specified. Otherwise, this specification can be omitted, and control returns to the statement immediately following the CALL.

The total number of arguments, $p_1,...,p_n + b_1,...,b_m$, must not exceed 63.

Actual arguments may be: arithmetic or logical expressions, constants, variables, array elements, array names, library function names, subroutine subprogram names, external function names (not an intrinsic or statement function), function references (the function reference is a special case of an arithmetic expression), or LEVEL 3 array names or variables.

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
```

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA(X,12)
      CALL IOTA(Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (*1ARRAY X = *,12F6.0,5X,*ARRAY Y = *6F6.0)
      STOP
      END
      SUBROUTINE IOTA (A,M)
C     IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
C     STARTING AT 1
      DIMENSION A(M)
      DO 1 I = 1,M
1     A(I)=I
      RETURN
      END
```

If a CALL is the last statement in a DO loop, looping continues until the DO loop is satisfied.

Example:

Calling Program

```
  DO 5 I = 1,20
  .
  .
5 CALL GRATER (STACK(I),TEMP(I))
  .
  .
```

Subprogram

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

The subroutine subprogram GRATER will be called 20 times.

Example:

Calling Program

```
.
.
DIMENSION LIST (50)
.
.
CALL SORT (LIST)
.
.
.
```

Subprogram

```
    SUBROUTINE SORT(ALIST)
    INTEGER ALIST (50)
    DO 10 J = 1,50
    K = 50 - J
    DO 10 I = 1,K
    IF (ALIST (I) - ALIST (I+1)) 15,10
 15 ITEMP = ALIST (I)
    ALIST (I) = ALIST (I + 1)
    ALIST (I + 1) = ITEMP
 10 CONTINUE
 50 WRITE (6,200) ALIST
200 FORMAT (*1*,10(I4,2X))
    RETURN
    END
```

The parameter list in a SUBROUTINE subprogram is optional.

Example:

Calling Program **Subprogram**

```
   .                                   SUBROUTINE ERROR1
   .                                   WRITE (6,1)
   .                                 1 FORMAT (5X,*NUMBER IS OUT OF RANGE*)

   IF (A-B) 10,20,20                   RETURN
   .                                   END
   .
   .
10 CALL ERROR1
20 RESULT=(A*CAT) +375.2-ZERO
   .
   .
   .
```

SUBPROGRAMS AND COMMON

Transferring values through common is a more efficient method of passing values than through arguments in the CALL statement. Variables or arrays in a calling program or a subprogram can share the same storage locations with variables or arrays in other subprograms. Therefore, a block of common storage can be used to transfer values between a calling program and a subprogram.

Example:

```
      PROGRAM CMN (INPUT,OUTPUT)
      COMMON NED (10)
      READ 3,NED
    3 FORMAT (10I3)
      CALL JAVG
      STOP
      END
      SUBROUTINE JAVG
C THIS SUBROUTINE COMPUTES THE AVERAGE OF THE FIRST 10 ELEMENTS IN
C  COMMON
      COMMON N(10)
      ISTORE = 0
      DO 1 I = 1,10
   1  ISTORE = ISTORE + N(I)
      ISTORE = ISTORE/10
      PRINT 2,ISTORE
    2 FORMAT (*1AVERAGE = *,I10)
      RETURN
      END

      AVERAGE =          45
```

The array NED in program CMN and the array N in subroutine JAVG share the same locations in common. NED(1) shares the same location with N(1), NED(2) with N(2), etc. The values read into locations NED(1) through NED(10) are available to subroutine JAVG. JAVG computes and prints the average of these values.

Arguments passed in COMMON are subject to the same rules with regard to type, length, etc., as those passed in an argument list (section 5).

ADJUSTABLE DIMENSIONS IN SUBPROGRAMS

Within a subprogram, array dimension specifications may use integer variables instead of constants, provided the array name and all integer names used for array dimension specifications are dummy arguments of the subprogram. The actual array name and values for the dummy variables are given by the calling program when the subprogram is called. The dimensions of a dummy array in a subprogram are adjustable and may change each time the subprogram is called; however, the absolute dimensions of the array must have been declared in a calling program. The size of an array passed to a subprogram using adjustable dimensions should not exceed the absolute dimensions of that array.

Adjustable dimensions cannot be used for arrays which are in common.

Calling Program

```
      DIMENSION A(10,10),B(10,10),C(10,10),
     S     E(5,5),F(5,5),G(5,5),H(10,10)
      .
      .
      .
      CALL MATADD (E,F,G,5,5)
      .
      .
      .
      CALL MATADD(A,B,C,10,7)
      CALL MATADD(B,C,A,I,10)
```

SUBROUTINE Subprogram

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

When this call is made to the subprogram, the actual arguments (A,B,C,10,7) are substituted for MATADD(X,Y,Z,M,N), and the subprogram is assigned dimensions: DIMENSION X(10,7),Y(10,7),Z(10,7)

The main program may call the subroutine MATADD from several places within the main program.

The adjustable dimensions may be passed through more than one level of subprograms.

Example:

Calling Program	Subprogram
.	SUBROUTINE SUB3 (B,I,J)
.	DIMENSION B(I,J)
.	.
REAL A(10,5)	.
CALL SUB3 (A,5,3)	.
.	DO 20 K = 1, J
.	.
.	CALL SUB4 (B,I,J)
	.
	.

Subprogram

```
SUBROUTINE SUB4 (X,K,L)
DIMENSION X (K,L)
.
.
```

In the main program, array A has dimensions (10,5); a portion of this array is passed to the subroutine SUB3 through the call CALL SUB3(A,5,3). Thus array B in the subroutine has dimensions (5,3). The subroutine SUB3, in turn, calls another subroutine SUB4 passing the dimensions of the array B. The array X in the subroutine SUB4 has dimensions X (5,3).

Constants must be used when array A is dimensioned in the initial calling program, and the values of second and third arguments in the subprogram call should be consistent with the dimensions of A. If adjustable dimensions are not consistent with constant dimensions in the calling program, results are undefined.

In a subprogram, an array name which appears in a COMMON statement must not have adjustable dimensions.

Example:

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA(X,12)
      CALL IOTA(Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (*1ARRAY X = *,12F6.0,5X,*ARRAY Y = *6F6.0)
      STOP
      END
      SUBROUTINE IOTA (A,M)
C     IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
C     STARTING AT 1
      DIMENSION A(M)
      DO 1 I = 1,M
1     A(I)=I
      RETURN
      END
```

ENTRY STATEMENT

```
          7
          ENTRY name
```

A subroutine or function subprogram may be entered at a point other than the first executable statement through the ENTRY statement.

In the subprogram, name may appear only in the ENTRY statement. The first executable statement following ENTRY becomes an alternate entry point to the subprogram. An ENTRY statement in a main program is ignored, and a warning diagnostic is printed.

Example:

```
Main Program              Subroutine Subprogram

COMMON SET1 (25)              SUBROUTINE CLEAR (ARAY)
.                             DIMENSION ARAY (25)
.                             DO 100 I = 1,25        Entry Point
CALL CLEAR (SET1)         100 ARAY (I) = 0.0
.                             ENTRY FILL
.                           3 READ 2, VALUE, IPLACE  Entry Point
.                           2 FORMAT (10X, F7.2, I4)
CALL FILL                     ARAY (IPLACE) = VALUE
.                             IF (IPLACE .GT. 24) RETURN
.                             GO TO 3
.                             END
```

At some point in the main program, a call is made to the subroutine: CALL CLEAR (SET1)

The array SET1 is set to zero and values are read into the array. Later in the program, a call is made again to the subroutine CLEAR; but this time it is entered at the entry point FILL: CALL FILL

When FILL is called, further values are read into the array SET1 without first setting the array to zero.

Each ENTRY name must appear in a separate ENTRY statement. ENTRY statements should not have statement labels; if a statement label appears, it is ignored and a warning diagnostic is printed. The ENTRY statement does not have any arguments; the dummy arguments appearing with the FUNCTION or SUBROUTINE statement are implied. A subroutine or function subprogram can contain any number of ENTRY statements.

The ENTRY statement may appear anywhere in the subprogram except within a DO loop. Within a DO loop, it is ignored and a warning diagnostic is printed. The ENTRY statement is referenced in the same way a SUBROUTINE or FUNCTION is referenced.

Example:

Main Program

```
Z=A+B-JOE(3.*P,Q-1)
.
.
.
R=S+JAM(Q,2.5*P)
.
.
.
```

Function Subprogram

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

In the calling program, an entry name may appear in an EXTERNAL statement, and FUNCTION entry names also may appear in type statements. All ENTRY points within a SUBROUTINE subprogram define SUBROUTINE subprogram names, and all ENTRY points within a FUNCTION subprogram define FUNCTION subprogram names. A function entry name assumes the same type as the name in the FUNCTION statement.

Example:

Main Program

```
REAL JOHN
.
.
R=A+JOHN(8,7)
.
.
```

Function Subprogram

```
   FUNCTION TED (X,Y)
   DO 20N = 1,10
20 TED = Y+3.7*X
   ENTRY JOHN
   READ 4, NUM, IFOR
 4 FORMAT (5X,6I6)
   JOHN = NUM+IFOR*20
   RETURN
   END
```

The name of the FUNCTION is implicitly typed real, therefore, JOHN assumes the type real. JOHN should be declared real in the calling program.

An ENTRY name must be unique in the FUNCTION subprogram.

Example:

```
FUNCTION CAT(A,B)
.
.
.
DOG=10.+3.2
ENTRY DOG
```

The ENTRY name DOG is not valid because it has been used as a variable.

The value of the function is the last value assigned to the name of the function regardless of which ENTRY statement was used to enter the subprogram. The function name is used to return results to the calling program even though the reference was through an entry name.

Example:

Calling Program	**Subprogram**
`RESULT=FSHUN(X,Y,Z)`	`FUNCTION FSHUN(A,B,C)`
`RES2=FRED(R,S,T)`	`3 FSHUN=A*B/C**2`
	`RETURN`
	`ENTRY FRED`
	`IF(A .LE. 702.) GO TO 3`
	`FSHUN=(C+A)/B`
	`RETURN`
	`END`

When the FUNCTION FSHUN is entered at the beginning of the function, or through the ENTRY FRED, the result must be returned to the calling program through the function name FSHUN.

Example:

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO 2 I = 1,M
    2 A(I) = A(I) + V
      RETURN
      END
```

An explanation of this example appears in part 2.

Certain subprograms that are of general utility or difficult to express in FORTRAN are supplied by the system. These library subprograms are referenced with a CALL statement or a function reference in exactly the same way a user written subprogram is referenced. The library provides:

External Functions. When an external function is referenced in a source program, a call is made to a library function, and the required result is returned to the user's program.

Intrinsic Functions. When the user references an intrinsic function, the compiler inserts code in-line in the object code of the source program unless the T, D, or OPT=0 option is selected on the FTN control card, or the name of the function appears in an EXTERNAL statement.

Utility Subprograms. In addition to the basic external functions and intrinsic functions specified by ANSI, the FORTRAN library contains additional utility subprograms for the user's convenience. Utility subprograms are always called by name (refer to section 7).

When a user defined external function has the same name as a library **intrinsic** function, the library function is referenced and the user defined function is ignored. However, a user defined function with the same name as a library **external** function overrides the library function.

INTRINSIC FUNCTIONS

If an intrinsic function name is used in any context other than a function reference, the user definition overrides the system definition. An intrinsic function name may be used as:

Variable

Array name

Statement function name

Name of a FUNCTION subprogram – only if it appears in a type statement of a different type or in an EXTERNAL statement.

Intrinsic function names should not be used for a user defined function name (unless they appear in an EXTERNAL or type statement). Otherwise, the user defined function will be ignored. (Refer to section 7).

If the control card options T, D, or OPT=0 are specified, intrinsic functions are called by name.

A list of the intrinsic functions defined in ANSI and the additional intrinsic functions provided by FORTRAN Extended follows. Non-ANSI intrinsic functions appear in blue type.

If a function has no mode, the result of the function assumes the type of the expression in which it is used. The 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 for +0 and negative for -0. The functions AMOD and MOD are not defined, however, when the second argument is zero; division by zero renders the results undefined.

Table 8-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Absolute Value	|A|	1	ABS IABS DABS	Real Integer Double	Real Integer Double	Y=ABS(X) J=IABS(I) DOUBLE A,B B=DABS(A)
Truncation	Sign of A times largest integer ≤ |A|	1	AINT INT IDINT	Real Real Double	Real Integer Integer	Y=AINT(X) I=INT(X) DOUBLE Z J=IDINT(Z)
Remaindering † (see note)	A1 (mod A2)	2	AMOD MOD	Real Integer	Real Integer	B=AMOD(A1,A2) J=MOD(I1,I2)
Choosing largest value	Max(A1, A2,...)	≥2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double	X=AMAX0(I,J,K) A=AMAX1(X,Y,Z) L=MAX0(I,J,K,N) I=MAX1(A,B) DOUBLE W,X,Y,Z W=DMAX1(X,Y,Z)
Choosing smallest value	Min(A1, A2,...)	≥2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double	Y=AMIN0(I,J) Z=AMIN1(X,Y) L=MIN0(X,Y) J=MIN1(X,Y) DOUBLE A,B,C C=DMIN1(A,B)
Float	Conversion from integer to real	1	FLOAT	Integer	Real	X1=FLOAT(I)

† MOD or AMOD (x1,x2) is defined as x1-|x1/x2| x2, where |x| is the largest integer that does not exceed the magnitude of x with sign the same as x.

Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Fix	Conversion from real to integer	1	IFIX	Real	Integer	IY=IFIX(Y)
Transfer of Sign	Sign of A2 with $\lvert A1 \rvert$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double	Z=SIGN(X,Y) J=ISIGN(I1,I2) DOUBLE X,Y,Z Z=DSIGN(X,Y)
Positive Difference	If A1>A2 then A1-A2. If A1 ≤A2 then 0.	2	DIM IDIM	Real Integer	Real Integer	A=DIM(C,D) J=IDIM(I1,I2)
Logical Product	Bit-by-bit logical AND of A_1 through A_n	$n \geq 2$	AND	any type ††	no mode	C=AND(A1,A2)
Logical Sum	Bit-by-bit logical OR of A_1 through A_n	$n \geq 2$	OR	any type ††	no mode	D=OR(A1,A2)
Exclusive OR	Bit-by-bit Exclusive OR of A_1 through A_n	$n \geq 2$	XOR	any type ††	no mode	D=XOR(A1,A2)
Complement	Bit-by-bit Boolean complement of A	1	COMPL	any type ††	no mode	B=COMPL(A)

†† For a double precision or complex argument, only the high order or real part will be used.

Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Shift	Shift A1, A2 bit positions: left circular if A2 is positive; right with sign extension, and end off if A2 is negative. If A2 is not a constant, with A2 $<$ 0, and $\|A2\|$ $>$ 63, the result is +0.	2	SHIFT	A1: any type †† A2:integer	no mode	B=SHIFT(A,I)
Mask	Form mask of A1 bits set to 1 starting at the left of the word. $0 \leqslant A1 \leqslant 60$	1	MASK	Integer	no mode	A=MASK(B)
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real	DOUBLE Y X=SNGL(Y)
Obtain Real Part of Complex Argument		1	REAL	Complex	Real	COMPLEX A B=REAL(A)

†† For a double precision or complex argument, only the high order or real part will be used.

Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real	COMPLEX A D=AIMAG(A)
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double	DOUBLE Y Y=DBLE(X)
Express Two Real Arguments In Complex Form	A1+A2i (where $i^2 = -1$)	2	CMPLX	Real	Complex	COMPLEX C C=CMPLX(A1,A2)
Obtain Conjugate of a Complex Argument	a-bi (where A=a+bi)	1	CONJG	Complex	Complex	COMPLEX X,Y Y=CONJG(X)

EXTERNAL FUNCTIONS

In the following list of basic external functions defined in ANSI and the additional external functions provided by FORTRAN Extended, non-ANSI functions appear in blue type.

External functions are called by value if the control card option T, D or OPT=0 is not specified and they do not appear in an EXTERNAL statement.

If the user defines a function with the same name as a library external function, his function overrides the library function. The name still retains the type of the library external function, however, and it will be called by value (if the control card option T, D, or OPT=0 is not selected) unless declared EXTERNAL. Therefore, if a user defined function has the same name as a library function, that name should appear in a type or EXTERNAL statement. (Refer to section 7 for an explanation of call-by-value).

In the following tables, A represents the argument. When more than one argument is involved, A1 is the first argument and A2 the second. 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.

Table 8-2. Basic External Functions

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Exponential	e**A	1 1 1	EXP DEXP CEXP	Real Double Complex	Real Double Complex	Z=EXP(Y) DOUBLE X,Y Y=DEXP(X) COMPLEX A,B B=CEXP(A)
Natural Logarithm	$\log_e(A)$	1 1 1	ALOG DLOG CLOG	Real Double Complex	Real Double Complex	Z=ALOG(Y) DOUBLE X,Y Y=DLOG(X) COMPLEX A,B B=CLOG(A)
Common Logarithm	$\log_{10}(A)$	1	ALOG10 DLOG10	Real Double	Real Double	B=ALOG10(A) DOUBLE D,E E=DLOG10(D)
Trigono-metric Sine	sin (A)	1 1 1	SIN DSIN CSIN	Real Double Complex	Real Double Complex	Y=SIN(X) DOUBLE D,E E=DSIN(D) COMPLEX CC,F CC=CSIN(CD)
Trigono-metric Cosine	cos (A)	1 1 1	COS DCOS CCOS	Real Double Complex	Real Double Complex	X=COS(Y) DOUBLE D,E E=DCOS(D) COMPLEX CC,F CC=CCOS(F)
Hyperbolic Tangent	tanh (A)	1	TANH	Real	Real	B=TANH(A)

Table 8-2. Basic External Functions (Continued)

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Square Root	$(A)^{1/2}$	1 1 1	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex	Y=SQRT(X) DOUBLE D,E E=DSQRT(D) COMPLEX CC,F CC=CSQRT(F)
Arctangent	arctan (A) arctan (A1/A2)	1 1 2 2	ATAN DATAN ATAN2 DATAN2	Real Double Real Double	Real Double Real Double	Y=ATAN(X) DOUBLE D,E E=DATAN(D) B=ATAN2(A1,A2) DOUBLE D,D1,D2 D=DATAN2(D2,D2)
Remaindering †	A1 (mod A2)	2	DMOD	Double	Double	DOUBLE DM,D1,D2 DM=DMOD(D1,D2)
Modulus	a^2+b^2 for A=a+bi	1	CABS	Complex	Real	COMPLEX C CM=CABS(C)
Arccosine	arccos (A)	1	ACOS	Real	Real	X=ACOS(Y)
Arcsine	arcsin (A)	1	ASIN	Real	Real	X=ASIN(Y)
Trigonometric Tangent	tan (A)	1	TAN	Real	Real	X=TAN(Y)

† The function DMOD (x1,x2) is defined as x1-|x1/x2| x2, where |x| is the largest integer that does not exceed the magnitude of x with sign the same as x.

ADDITIONAL UTILITY SUBPROGRAMS

The following utility subroutines are supplied by the system. ANSI does not specify any library subroutines.

A user supplied subprogram with the same name as a library subprogram overrides the library subprogram, but still retains the type of the library subprogram.

The subprograms which follow are always called by name (refer to section 7).

In the following definitions, i is an integer variable or constant; j is an integer variable.

SUBROUTINES

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

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

Dumps central memory (small core storage) on the OUTPUT file in the indicated format. If PDUMP was called, it returns control to the calling program; if DUMP was called, it terminates program execution. a_i identifies the first word and b_i the last word of the storage area to be dumped; $1 \leq n \leq 20$. f is a format indicator, as follows:

f = 0 or 3, octal dump

f = 1, real dump

f = 2, integer dump

f = 4, octal dump

a and b indicate the range of addresses to dump rather than specific addresses, therefore, if statement labels are specified, an ASSIGN statement must be used to define a and b. A dump begins at the statement assigned to a_1 and ends with the statement number assigned to b_1.

The maximum number of arguments is 63.

Example: `CALL PDUMP (A,B,0,M,N,4,X,X(100),1)`

CALL SSWTCH (i,j)

If sense switch i is on, j is set to 1; if sense switch i is off, j is set to 2. i is 1 to 6. If i is out of range, an informative diagnostic is printed, and j = 2. The computer operator uses this subroutine to select options in a FORTRAN program.

CALL REMARK (H)

Places a message of not more than 40 characters, in the dayfile. H is a Hollerith specification.

Example: `CALL REMARK (9HLAST DECK)`

CALL DISPLA (H,k)

Displays a name and a value in the dayfile. H is a Hollerith specification of not more than 40 characters, k is a variable, or a real or integer expression; k is displayed as an integer or real value.

Example: `CALL DISPLA (7H TIME =, STOP-START)`

CALL RANGET(n)

Obtains current generative value of RANF between 0 and 1. n is a symbolic name to receive the seed. It is not normalized.

CALL RANSET(n)

Initializes generative value of RANF. n is a bit pattern. Bit 2^0 will be set to 1 (forced odd), and bits (2^{59}-2^{48}) will be set to 1717 octal.

SECOND(t) or CALL SECOND (t)†

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

Example: `DPTIM = SECOND (CP)`

DATE(a) or CALL DATE (a)†

The value of a will be the current date in operating system format. a is a dummy argument. Format is bMM/DD/YYb; but it may vary at installation option.

Examples:

```
TODAY = DATE (D)

CALL DATE (TODAY)
```

†These routines can be used as functions or subroutines. The value is always returned via the argument and the normal function return.

TIME(a) or **CALL TIME (a)**†

The value of a will be the current reading of the system clock. Format is HH.MM.SS.

Examples:

```
TYM = TIME (C)

CALL TIME (C)
```

CALL ERRSET (a,b)††

Sets maximum number of errors, b, allowed in input data before fatal termination. Error count is kept in a.

CALL LABEL (u,fwa)††

Sets tape label information for a file. u is the unit number. fwa is the address of the first word of the label information. The label information must be in the mode and format discussed in the SCOPE Reference Manual.

CALL MOVLEV (a,b,n)

Transfers n consecutive words of data between a and b. a and b are variables or array elements; n is an integer constant or expression. a is the starting address of the data to be moved and b is the starting address of the location to receive it.

Example:

```
CALL MOVLEV(A,B,1000)
```

CALL OPENMS (u,ix,lngth,t)†††

Opens mass storage file and informs Record Manager that this file is word addressable. If an existing file is called, the master index is read into the area specified by the program. u is the unit designator. ix is the first word address of the index in central memory. lngth is the length of the index buffer; for a name index, lngth $\geq$ 2 * (number of records in file) + 1; for a number index, lngth $\geq$ number of records in file + 1. t = 1 file is referenced through a name index; t = 0 file is referenced through a number index.

Example:

```
PROGRAM MS1 (TAPE3)
DIMENSION INDEX (11), DATA (25)
CALL OPENMS (3,INDEX,11,0)
```

---

†These routines can be used as functions or subroutines. The value is always returned via the argument and the normal function return.

††Refer to section 5, part 3 for further information.

†††Refer to section 7, part 3 for further information.

CALL READMS (u,fwa,n,k)†

Transmits data from mass storage to central memory. fwa is the central memory address of the first word of the record. n is the number of central memory words transferred. Number index k = $1 \leq k \leq$ lngth - 1. Name index k = any 60-bit quantity except ± 0. u is the unit designator.

Example: `CALL READMS(3,DATA,25,6)`

CALL WRITMS(u,fwa,n,k,r,s)†

Transmits data from central memory to mass storage. u,fwa,n,k are the same as for READMS. r = +1 rewrites in place. Unconditional request; fatal error is printed if new record length exceeds old record length. r = -1 rewrites in place if space is available, otherwise writes at end of information. r = 0 no rewrite; writes normally at end of information. The r parameter can be omitted if the s parameter is omitted. The default value for r is 0 (normal write).

s = 1 writes subindex marker flag in index control word for this record. s = 0 does not write subindex marker flag in index control word for this record. The s parameter can be omitted; its default value is 0.

The s parameter is included for future random file editing routines. Current routines do not test the flag, but the user should include this parameter in new programs, when appropriate, to facilitate transition to a future edit capability.

Example: `CALL WRITMS (3,DATA,25,NRKEY)`

CALL STINDX (u,ix,lngth,t)†

Changes index in central memory from master to subindex. u,ix,lngth,t are the same as OPENMS.

Example: `CALL STINDX (2,SUBIX,10)`

CALL CLOSMS (u)†

Writes index from central memory to file and closes file.

Example: `CALL CLOSMS (7)`

CALL STRACE

Provides subroutine calling traceback information from the subroutine which calls STRACE back to the main program. Traceback information is written to the file DEBUG. To obtain traceback information interspersed with the source program, DEBUG should be equivalenced to OUTPUT in the PROGRAM statement. (Refer to section 11, STRACE).

---

†Refer to section 7, part 3 for further information.

FUNCTIONS

RANF (n)

Random number generator. Returns values uniformly distributed over the range [0,1). n is a dummy argument which is ignored.

LOCF(a)

Returns address of argument a, which can be any type. Result is type integer.

UNIT (u)

Returns buffer status on unit u. Result is type real. -1 Unit ready, no error. +0 Unit ready, EOF encountered. + 1 Unit ready, parity error encountered.

Example: `IF (UNIT(2))30,40,70`

EOF(u)†

Gives input/output status on non-buffer unit. If zero, no end-of-file was encountered on previous read. Result is type real.

Example: `IFL = EOF (4)`

LENGTH(u)†

Gives number of central memory words read on the previous buffer or mass storage input/output request for a designated file. Result is type integer.

Example: `CMW = LENGTH(5)`

IOCHEC(u)†

Gives parity status on non-buffer unit. If zero, no parity error occurred on previous read.

LEGVAR(a)

Checks variable a. Result is -1 if variable is indefinite, + 1 if out of range, and 0 if normal. Variable a is type real; result is type integer.

---

†Refer to section 5, part 3 for further information.

The following subroutines are included for compatibility with previous processors only and should be avoided by new programs.

CALL FTNBIN (i,m,IRAY)

Null routine. All parameters ignored. Exists only for compatibility reasons.

CALL SLITE(i)

Turns on sense light i. If i = 0, turn all sense lights off. If i is other than (0-6), an informative diagnostic is printed; and sense lights are not changed.

CALL SLITET(i,j)

Tests sense light. If sense light i is on, j = 1, if sense light i is off, j = 2. Always turns sense light i off. If i is other than 1-6, an informative diagnostic is printed; all sense lights remain unchanged; and j = 2.

(Note: Logical variables generally provide a more efficient method of testing a condition than do calls to SLITE or SLITET).

CALL EXIT

Terminates program execution and returns control to the operating system.

CALL WRITEC (a,b,n)

Transfers data from central memory to extended core storage or LCM.

CALL READEC (a,b,n)

Transfers data from extended core storage or LCM to central memory. a is a simple variable or array element located in central memory, b is a simple variable or array element located in an extended core storage block or LCM. n is an integer constant or expression. n consecutive words of data are transferred beginning with a in central memory and b in extended core storage.

Examples:

```
      PROGRAM PIE(OUTPUT,TAPE6=OUTPUT)
      DATA CIRCLE,DUD/2*0.0/
      NAMELIST/OUT/PI

      DO 1 I = 1,10000
      X=RANF(DUD)
      Y=RANF(DUD)
      IF(X*X+Y*Y.LE.1.)CIRCLE=CIRCLE+1.
1     CONTINUE

      PI=4.*CIRCLE/10000.
      WRITE(6,OUT)

      STOP
      END
```

```
      PROGRAM TP (TAPE1,OUTPUT)
      INTEGER REC(512),RNUMB
      REWIND 1
      DO 4 RNUMB = 1,10000

    1 BUFFER IN (1,1) (REC(1),REC(512))

    2 IF (UNIT(1)) 3,5,5
    3 K=LENGTH(1)

C LENGTH RETURNS THE NUMBER OF WORDS IN CENTRAL MEMORY

    4 PRINT 100,RNUMB,(REC(I),I=1,K)
  100 FORMAT (7H0RECORD,I5/(1X,10A10))
    5 STOP
      END
```

```
      PROGRAM LOGIC(INPUT,OUTPUT,TAPE5=INPUT)
      LOGICAL MALE,PHD,SINGLE,ACCEPT
      INTEGER AGE
      PRINT 20
   20 FORMAT (*1                    LIST OF ELIGIBLE CANDIDATES*)
    3 READ (5,1) LNAME,FNAME,MALE,PHD,SINGLE,AGE
    1 FORMAT (2A10,3L5,I2)
      IF (EOF(5))6,4
    4 ACCEPT = MALE .AND. PHD .AND. SINGLE .AND. (AGE .GT. 25 .AND.
     S     AGE .LT. 45)
      IF (ACCEPT) PRINT 2,LNAME,FNAME,AGE
    2 FORMAT (1H0,2A10,3X,I2)
      GO TO 3
 6    STOP
      END
```

```
      PROGRAM LIBS (OUTPUT)
C
      EXTERNAL DATE
C
      CALL DATE (TODAY)
      CALL TIME (CLOCK)

      PRINT 2, TODAY, CLOCK
    2 FORMAT(*1TODAY=*, A10, * CLOCK=*, A10)
C
      CALL SECOND(TYME)
      LOCATN=LOCF(DATE)
      CALL RANGET(SEED)

      PRINT 3,TYME, LOCATN, LOCATN, SEED, SEED
      .
      .
      .
```

INPUT/OUTPUT I-9

To input or output data, the following information is required:

Unit number of the input/output device

List of FORTRAN variables to receive input data or from which results are to be output.

Layout or format of data

READ or WRITE statements specify the device and the list. The form of data is designated by the FORMAT statement.

Data can be formatted or unformatted. In formatted mode, display code character strings are converted and transferred according to a FORMAT statement. In unformatted mode, data is transferred in the form in which it normally appears in storage, no conversion takes place, and no FORMAT statement is used.

Input/output control statements are discussed below. Input/output lists and the FORMAT statements are covered in section 10.

The following definitions apply to all input/output statements:

u — Input/output unit; the SCOPE operating system associates this unit with an internal file name which may be:

Integer constant of one or two digits (leading zeros are discarded). The compiler associates these numbers with file names of the type TAPEu, where u is the file designator (refer to PROGRAM statement, section 7).

Simple integer variable name with a value of:

1 - 99, or

A display code file name (L format, left justified with zero fill). This is the internal logical file name.

fn — Format designator; a FORMAT statement number or the name of an array containing the format specification. The statement number must identify a FORMAT statement in the program unit containing the input/output statement.

iolist — Input/output list specifying items to be transmitted (section 10).

Under the SCOPE operating system, all information is considered to be a file or part of a file. Local to a given job, a file is identified by a logical file name (the internal file named, u). All control card references to a file identify it by the logical file name. The internal central memory representation of a logical file name consists of its literal value in display code, left justified and zero filled.

SCOPE treats several file names as special cases. When one of these names is used, SCOPE makes the following automatic dispositions, unless the user has defined an alternate disposition:

Card input is assigned to the file INPUT.

Data in the file OUTPUT is assigned to the printer.

Data in the file PUNCH is assigned to the card punch as coded card output.

Data in the file PUNCHB is output on the card punch as binary card output.

FORTRAN RECORD LENGTH

Formatted logical records can be a maximum of 150 characters for input, and 137 characters for output. The maximum length formatted logical records for cards is 80 characters.

The length of an unformatted FORTRAN logical record is determined by the length of the input/output list, and can be any size.

CARRIAGE CONTROL

In SCOPE output files assigned to the printer, a maximum of 137 characters can be specified for a line, but only 136 characters are printed. The first character of a line is the carriage control; it is never printed. The second character in the line appears in the first print position. The printer control characters are listed in section 10. For off-line printing, printer control is determined by the installation printer routine.

If more than 137 characters are specified for a line, a fatal execution time error results and an error message is printed.

OUTPUT STATEMENTS

PRINT

```
      PRINT fn,iolist
```

7

```
      PRINT fn
```

This statement transfers information from the storage locations named in the input/output list to the file named OUTPUT, according to the specification in the format designator, fn. If the user has not specified an alternate assignment, the file OUTPUT is sent to the printer.

```
      PROGRAM PRINT (OUTPUT)
      A=1.2
      B=3HYES
      N=19
      PRINT 4,A,B,N
    4 FORMAT (G20.6,A10,I5)
```

The iolist can be omitted. For example,

```
      PRINT 20
   20 FORMAT (30H THIS IS THE  END OF THE REPORT)
```

PUNCH

```
      PUNCH fn,iolist
```

```
      PUNCH fn
```

Data is transferred from the storage locations specified by iolist to the SCOPE file PUNCH. If the user has not specified an alternate assignment, the file PUNCH is output on the standard punch unit as Hollerith codes, 80 characters or less per card in accordance with format specification. fn. If the card image is longer than 80 characters, a second card is punched with the remaining characters.

```
      PROGRAM PUNCH (INPUT,OUTPUT,PUNCH)
    2 READ 3,A,B,C
    3 FORMAT (3G12.6)
      ANSWER = A + B - C
      IF (A .EQ. 99.99) STOP
      PRINT 4, ANSWER
    4 FORMAT (G20.6)
      PUNCH 5,A,B,C,ANSWER
    5 FORMAT (3G12.6,G20.6)
      GO TO 2
      END
```

The iolist can be omitted. For example,

```
      PUNCH 30
   30 FORMAT (10H LAST CARD)
```

FORMATTED WRITE

```
|     7
|     WRITE (u,fn) iolist
```

```
|     7
|     WRITE (u,fn)
```

Data is transferred from storage locations specified by iolist to the unit u according to FORMAT declaration, fn.

```
      PROGRAM RITE (OUTPUT,TAPE6=OUTPUT)
      X=2.1
      Y=3.
      M=7
      WRITE (6,100) X,Y,M
  100 FORMAT (2F6.2,I4)
      STOP
      END
```

The iolist can be omitted. For example,

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

UNFORMATTED WRITE

```
|     7
|     WRITE (u) iolist
```

Example:

```
      PROGRAM OUT(OUTPUT,TAPE10=OUTPUT)
      .
      .
      .
      DIMENSION A(260),B(4000)
      WRITE (10) A,B
      END
```

This statement is used to output binary records. Information is transferred from the list variables, iolist, to the specified output unit, u, with no FORMAT conversion. One record is created by an unformatted WRITE statement. (Refer to section 5, part 3). If the list is omitted, the statement writes a null record on the output device. A null record has no data but contains all other properties of a legitimate record.

INPUT STATEMENTS

FORMATTED READ

The user should test for an end-of-file after each READ statement to avoid input/output errors. If an attempt is made to read on unit u and an EOF was encountered on the previous read operation on this unit, execution terminates and an error message is printed. (Refer to section 5, part 3, EOF FUNCTION.)

```
      7
      READ fn,iolist
```

This statement transmits data from the INPUT file to the locations named in iolist. Data is converted in accordance with format specification fn.

```
      PROGRAM RLIST (INPUT,OUTPUT)
      READ 5,X,Y,Z
    5 FORMAT (3G20.2)
      RESULT = X-Y+Z
      PRINT 100, RESULT
  100 FORMAT (10X,G10.2)
      STOP
      END
```

```
      READ (u,fn) iolist
```

```
      7
      READ (u,fn)
```

This statement transmits data from unit, u, to storage locations named in iolist according to FORMAT specification fn. The number of words in the list and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, READ (u,fn) spaces over one FORTRAN record (section 10, FORTRAN Record /) unless the H specification is used to read Hollerith characters into an existing H field within the format specification.

```
      PROGRAM IN (INPUT,OUTPUT,TAPE4=INPUT,TAPE7=OUTPUT)
      READ (4,200) A,B,C
  200 FORMAT (3F7.3)
      A = B*C+A
      WRITE (7,50) A
   50 FORMAT (50X,F7.4)
      STOP
```

UNFORMATTED READ

```
READ (u) iolist
```

```
      7
      READ (u)
```

One record (refer to section 5, part 3) of information is transmitted from the specified unit, u, to the storage locations named in iolist. Records must be in binary form; no format statement is used. The information is transmitted from the designated file in the form in which it exists on the file. If the number of words in the list exceeds the number of words in the record, excess words in the list retain their previous values. If iolist is omitted READ (u) spaces over one record. If the number of locations specified in the iolist is less than the number of words in the logical record, an execution diagnostic is printed.

```
PROGRAM AREAD (INPUT,OUTPUT,TAPE2=INPUT)
READ (2) X,Y,Z
SUM = X+Y+Z/2.
WRITE (10) SUM
END
```

FILE MANIPULATION STATEMENTS

```
      7
      REWIND u
```

The REWIND operation positions a file so that the next FORTRAN input/output operation references the first record in the file. A mass storage file is positioned at the beginning of information. If the file is already at beginning of information, the statement acts as a do-nothing statement. (Refer to BACKSPACE/REWIND, section 5, part 3 for further information.)

Example:

```
REWIND 3
```

```
      7
      BACKSPACE u
```

Unit u is backspaced one logical record. If the file is at beginning of information, this statement acts as a do-nothing statement. 7600 only: BACKSPACE is permitted for F, S, or W record format or tape files with one record per block. (Refer to BACKSPACE/REWIND, section 5, part 3 for further information.)

Example:

```
   DO 1 LUN = 1,10,3
 1 BACKSPACE LUN
```

Files TAPE1, TAPE4, TAPE7, and TAPE10 are backspaced one logical record.

```
7
ENDFILE u
```

An end-of-file mark is written on the designated unit.

Example:

```
   IOUT = 6LOUTPUT
   END FILE IOUT
```

End-of-file is written on the file OUTPUT.

Extended core storage and mass storage input/output statements are discussed in section 7, part 3.

BUFFER STATEMENTS

The buffer statements and the read/write statements both accomplish data input/output: however, they differ in the following respects:

A buffer control statement initiates data transmission and then returns control to the program so that it can perform other tasks while data transmission is in progress. A read/write statement completes data transmission before returning control to the program.

In a buffer control statement, parity must be specified by a parity indicator. In the read/write control statement, the mode of transmission formatted (display code) or unformatted (binary) is tacitly implied.

The read/write control statements are associated with a list and, if formatted, with a FORMAT statement. The buffer statements are not associated with a list: data is transmitted to or from a block of storage.

```
7
BUFFER IN (u,p) (a,b)
```

p — Integer constant or simple integer variable. Designates parity on 7-track magnetic tape, zero designates even parity; one designates odd parity. p is inoperative for other peripheral devices.

a First word of record to be transmitted.

b Last word of record to be transmitted. The address of b must be greater than the address of a. Arrays are stored in the order in which they appear in the dimension declaration with larger subscripts at higher addresses.

Information is transmitted from unit u in either formatted or unformatted mode to storage locations a through b. Between the time a BUFFER IN statement is executed and the time a UNIT function on the same unit indicates the buffer operation is complete, neither the unit u nor the contents of storage locations a through b must be referenced by any statement. Only one record (section 5, part 3) is read for each BUFFER IN statement. If buffer status is not checked, the result of the last buffer operation is unpredictable.

```
1    5 7
      PROGRAM TP (TAPE1,OUTPUT)
      INTEGER REC(512),RNUMB
      REWIND 1
      DO 4 RNUMB = 1,10000
    1 BUFFER IN (1,1) (REC(1),REC(512))
    2 IF (UNIT(1)) 3,5,5
    3 K=LENGTH(1)
C LENGTH RETURNS THE NUMBER OF WORDS IN CENTRAL MEMORY
    4 PRINT 100,RNUMB,(REC(I),I=1,K)
  100 FORMAT (7HORECORD,I5/(1X,10A10))
    5 STOP
      END
```

Odd parity information is input from logical unit 1 beginning at the first word of the record REC(1), and extending through the last word of the record REC(512). The IF UNIT statement tests the status of the buffer operation. If the buffer operation is completed without error, statement 3 is executed. If an EOF or a parity error is encountered, control transfers to statement 5 and the program stops.

Additional Example:

```
DIMENSION CALC(50)
BUFFER IN (1,0) (CALC(1),CALC(50))
```

Even parity information is input from logical unit 1 beginning at the first word of the record, CALC(1), and extending through CALC(50), the last word of the record.

```
      7
      BUFFER OUT (u,p) (a,b)
```

u,p,a,b are the same as for BUFFER IN

Contents of storage locations a through b are written on unit u in even or odd parity.

Examples:

```
BUFFER OUT(2,0)(OUTBUF(1),OUTBUF(4))

DIMENSION ALPHA(100)
BUFFER OUT (2,1)(ALPHA(1),ALPHA(100))
```

One record is written for each BUFFER OUT statement. Section 5, part 3 contains further information regarding BUFFER IN/OUT statements.

NAMELIST

The NAMELIST statement permits input and output of groups of variables and arrays with an identifying name. No format specification is used.

```
      7
      NAMELIST/group name₁ /a₁ ,...,aₙ/ .../group nameₙ/a₁ ,...,aₙ
```

group name	Symbolic name which must be enclosed in slashes and must be unique within the program unit.
$a_1,...,a_n$	List of variables, array elements, or array names separated by commas.

The NAMELIST group name identifies the succeeding list of variables or array names. Whenever an input or output statement references the NAMELIST name, the complete list of associated variables or array names is read or written.

A NAMELIST group name must be declared in a NAMELIST statement before it is used in an input/output statement. The group name may be declared only once, and it may not be used for any purpose other than a NAMELIST name in the program unit. It may appear in any of the input/output statements in place of the format number:

READ (u, group name)
READ group name
WRITE (u, group name)
PRINT group name
PUNCH group name

It may not, however, be used in an ENCODE or DECODE statement in place of the format number. When a NAMELIST group name is used, the list must be omitted from the input/output statement.

A variable, array element or array name may belong to one or more NAMELIST groups.

Data read by a single NAMELIST name READ statement must contain only names listed in the referenced NAMELIST group. A set of data items may consist of any subset of the variable names in the NAMELIST. The value of variables not included in the subset remain unchanged. Variables need not be in the order in which they appear in the defining NAMELIST statement.

```
PROGRAM NMLIST (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
NAMELIST/SHIP/A,B,C,I1,I2
READ(5,SHIP)
IF (C.LE.0.)STOP
A=B+C
I1=I2 + I1
WRITE (6,SHIP)
END
```

Input record

```
 2
$SHIP A=12.2,B=20.,C=3.4,I1=8,I2=50$
```

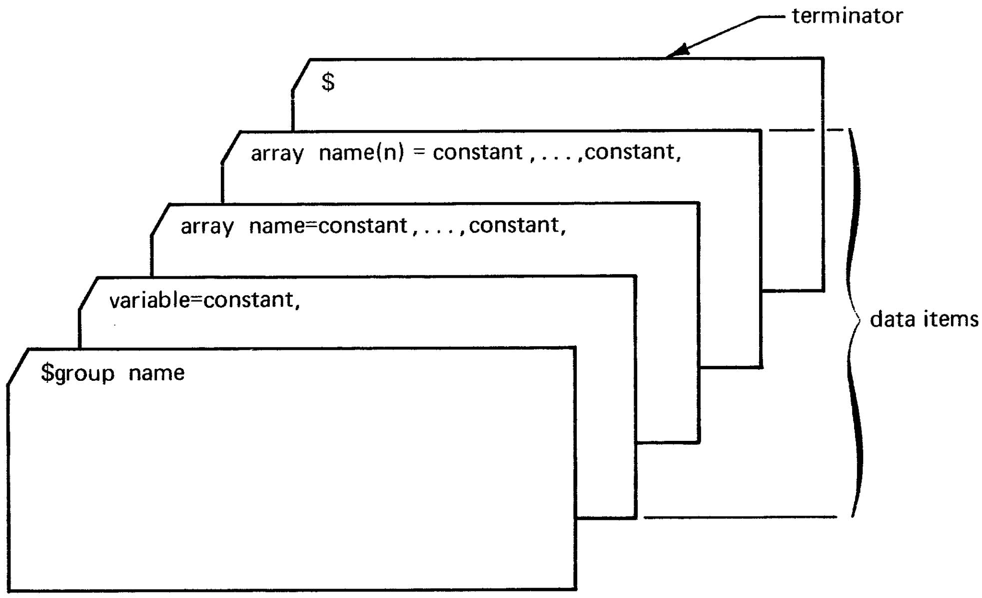
Output

```
$SHIP

A        =  0.234E+02,

B        =  0.2E+02,

C        =  0.34E+01,

I1       =  58,

I2       =  50,

$END
```

```
        7
        READ (u,group name)
```

When a READ statement references a NAMELIST group name, input data, in the format described below, is read from the designated unit, u. If no group name is found, and an end of file is encountered, a fatal error occurs.

INPUT DATA



Data items succeeding $ NAMELIST group name are read until another $ is encountered.

Blanks must not appear:

Between $ and NAMELIST group name

Within array names and variable names

Within constants and repeated constant fields

Blanks may be used freely elsewhere.

A maximum of 150 characters per input record is allowed. More than one record may be used in input data. The first column of each record is ignored. All except the last record must end with a constant followed by a comma.

Data items separated by commas may be in three forms:

variable = constant

array name = constant,....constant

array name(integer constant subscripts)=constant,....constant

Constants can be preceded by a repetition factor and an asterisk.

Example:

`5*(1.7,-2.4)` five complex constants.

Constants may be integer, real, double precision, complex or logical. Logical constants must be of the form: .TRUE. .T. T .FALSE. .F. or F. A logical variable may be replaced only by a logical constant. A complex variable may be replaced only by a complex constant. A complex constant must have the form (real constant, real constant). Any other variable may be replaced by an integer, real or double precision constant; the constant is converted to the type of the variable.

OUTPUT

```
7
WRITE(u,group name)
```

All variables and arrays, and their values, in the list associated with the NAMELIST group name are output on the designated unit, u. They are output in the order of specification in the NAMELIST Statement. Output consists of at least three records. The first record is a $ in column 2 followed by the group name; the last record is a $ in column 2 followed by the characters END.

Example:

```
PROGRAM NAME(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
NAMELIST/VALUES/TOTAL,QUANT,COST
DATA QUANT,COST/15.,3.02/
TOTAL = QUANT*COST*1.3
WRITE (6,VALUES)
STOP
END
```

Output

```
$VALUES

TOTAL   =  0.58889999999999E+02,

QUANT   =  0.15E+02,

COST    =  0.302E+01,

$END
```

No data appears in column 1 of any record. If the logical unit referenced is the standard punch unit and a variable crosses column 80, this and following variables are punched on the next card. The maximum length of a record written by a WRITE (u,group name) statement is 130 characters. Logical constants appear as T or F. Elements of an array are output in the order in which they are stored.

Records output by a WRITE (u, group name) statement may be read by a READ (u, group name) statement using the same NAMELIST name.

Example:

```
NAMELIST/ITEMS/X,Y,Z
.
.
.
WRITE (6,ITEMS)
```

Output record:

```
$ITEMS
X=734.2,
Y=2374.9,
Z=22.25,
$END
```

This output may be read later in the same program using the following statement:

```
READ(5,ITEMS)
```

ARRAYS IN NAMELIST

In input data the number of constants, including repetitions. given for an array name should not exceed the number of elements in the array.

Example:

```
DIMENSION BAT(10)
NAMELIST/HAT/BAT,DOT
READ (5,HAT)
```

```
 2
 $HAT        BAT=2,3,8*4,DOT=1.05$END
```

The value of DOT becomes 1.05, the array BAT is as follows:

BAT(1)	2
BAT(2)	3
BAT(3)	4
BAT(4)	4
BAT(5)	4
BAT(6)	4
BAT(7)	4
BAT(8)	4
BAT(9)	4
BAT(10)	4

Example:

```
DIMENSION GAY(5)
NAMELIST/DAY/GAY,BAY,RAY
READ (5,DAY)
```

Input Record:

```
 2
 $DAY GAY(3)=7.2,GAY(5)=3.0,BAY=2.3,RAY=77.2$
```

array element (integer constant subscript)=constant.....constant

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants need not equal, but may not exceed, the remaining number of elements in the array.

Example:

```
DIMENSION ALPHA (6)
NAMELIST/BETA/ALPHA,DELTA,X,Y
READ (5,BETA)
```

Input record:

```
 2
 $BETA ALPHA(3)=7.,8.,9.,DELTA=2.$
```

In storage

```
ALPHA(3)          7.
ALPHA(4)          8.
ALPHA(5)          9.
DELTA             2.
```

Data initialized by the DATA statement can be changed later in the program by the NAMELIST statement.

Example:

```
PROGRAM COSTS (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
DATA TAX,INT,ACCUM,ANET/23.,10,500.2,17.0/
NAMELIST/RECORDS/TAX,INT,ACCUM,ANET
FIRST = TAX + INT
SECOND = FIRST * SUM
.
.
.
READ(5, RECORDS)
.
.
.
```

Input Record:

```
 2
 $RECORDS TAX=27., ACCUM=666.2$
```

Example:

```
DIMENSION Y(3,5)
LOGICAL L
COMPLEX Z
NAMELIST/HURRY/I1,I2,I3,K,M,Y,Z,L
READ (5,HURRY)
```

Input Record:

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

produce the following values:

```
I1=1               Y(1,2)=14.0
I2=2               Y(2,2)=14.0
I3=3               Y(3,2)=14.0
Y(3,5)=26.0        Y(1,3)=14.0
Y(1,1)=11.0        K=16
Y(2,1)=120.0       M=17
Y(3,1)=13.0        Z=(1.,2.)
                   L=.TRUE.
```

ENCODE AND DECODE

The ENCODE and DECODE statements are used to reformat data in memory; information is transferred under FORMAT specifications from one area of central memory (small core) storage to another.

ENCODE is similar to a WRITE statement, and DECODE is similar to a READ statement. Data is transmitted under format specifications, but ENCODE and DECODE transfer data internally; no peripheral equipment is involved. For example, data can be converted to a different format internally without the necessity of writing it out on tape and rereading under another format.

ENCODE

```
      7
      ENCODE (c,fn,v) iolist
```

v — Variable or array name which supplies the starting location of the record to be encoded.

c — Unsigned integer constant or simple integer variable specifying the length of each record.

The first record starts with the leftmost character of the location specified by v and continues for c characters, 10 characters per computer word. If c is not a multiple of 10, the record ends before the end of the word is reached; and the remainder of the word is blank filled. Each new record begins with a new computer word. c must be less than or equal to 150.

fn — Format designator, statement label or integer variable, which must not be a NAMELIST group name.

iolist — List of variables to be transmitted to the location specified by v.

Example:

```
     5 7
       PROGRAM ENCDE (OUTPUT)
       DIMENSION A(2),ALPHA(4)
       DATA A,B,C/10HABCDEFGHIJ,10HKLMNOPQRST,5HUVWXY,7HZ123456/
       ENCODE (40,1,ALPHA)A,B,C
     1 FORMAT (2A4,A5,A6)
       PRINT 2,ALPHA
     2 FORMAT (20H1CONTENTS OF ALPHA =,8A10)
       STOP
       END
```

In memory after ENCODE statement has been executed.

ALPHA (1)	ALPHA (2)	ALPHA (3)	ALPHA (4)
ABCDKLMNUV	WXYZ12345		

ENCODE is a core-to-core transfer of data, which is similar to WRITE. Only data in central memory or small core can be transferred by the ENCODE statement. Data in the iolist, in internal form, is converted under FORMAT specifications, fn, and written in display code into an array or variable.

An integral number of words is allocated for each record created by an ENCODE statement. If c is not a multiple of 10, the record ends before the end of the word is reached; and the remainder of the word is blank filled.

The number of characters allocated for any single record in the encoded area must not exceed 150.

Example:

`10 FORMAT (16F10.4)`	illegal (length > 150 characters); fatal error and the message EXCEEDED RECORD SIZE is printed.
`10 FORMAT (10F10.4/6F10.4)`	legal

If the list and the format specification transmit more than the number of characters specified per record, an execution error message is printed. If the number of characters transmitted is less than the length specified by c, remaining characters in the record are blank filled.

For example, in the following program which is similar to program ENCDE above, the format statement has been changed; so that two records are generated by the ENCODE statement. A(1) and A(2) are written with the format specification 2A4, the / indicates a new record, and the remaining portion of the 40 character record, c, is blank filled. B and C are written into the second record with the specification A5 and A6, and the remaining characters are blank filled. The dimensions of the array ALPHA must be increased to 8 to accommodate two 40-character records.

```
      PROGRAM TWO (OUTPUT)
      DIMENSION A(2),ALPHA(8)
      DATA A,B,C/10HABCDEFGHIJ,10HKLMNOPQRST,5HUVWXY,7HZ123456/
      ENCODE (40,1,ALPHA)A,B,C
    1 FORMAT (2A4/A5,A6)
      PRINT 2,ALPHA
    2 FORMAT (20H1CONTENTS OF ALPHA =,8A10)
      STOP
      END
```

Output:

```
CONTENTS OF ALPHA =ABCDKLMN                                 UVWXYZ12345
```

If this same ENCODE statement is altered to:

```
  ENCODE (33,1,ALPHA)A,B,C
1 FORMAT (2A4/A5,A6)
```

The contents of ALPHA remain the same. When a record ends in the middle of a word the remainder of the word is blank filled (each new record starts at the beginning of a word).

Record 1				Record 2			
ABCDKLMN			blank fill	UVWXYZ1234	5		blank fill
ALPHA(1)	ALPHA(2)	ALPHA(3)	ALPHA(4)	ALPHA(5)	ALPHA(6)	ALPHA(7)	ALPHA(8)
			end of record				end of record

The array in core must be large enough to contain the total number of characters specified in the ENCODE statement. For example, if 70 characters are generated by the ENCODE statement, the array starting at location v (if v is a single word element) must be dimensioned at least 7. If 27 characters are generated, the array must be dimensioned 3. If only 6 characters are generated, v can be a 1-word variable.

The following example illustrates that it is possible to encode an area into itself, and the information previously contained in the area will be destroyed.

```
      PROGRAM ENCO2 (OUTPUT)
      I=10HBCDEFGHIJK
      IA=1H1
      ENCODE (8,10,I) I,IA,I
   10 FORMAT (A3,A1,R4)
      PRINT 11,I
   11 FORMAT (A11)
      END
```

Printout is:

BCD1HIJKbb

ENCODE 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. The following program permits m to vary in the range 2 through 9.

```
       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 ensure it is within limits; if it is 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, through I9 according to the value of m.

The following program is another example of forming FORMAT statements internally:

```
      PROGRAM IGEN (OUTPUT,TAPE6=OUTPUT)
      DO 9 J=I,50
      ENCODE (10,7,FMT)J
    7 FORMAT (2H(I,I2,1H))
    9 WRITE (6,FMT)J
      STOP
      END
```

In memory, FMT is first (I 1) then (I 2), then (I 3), etc.

DECODE

```
7
DECODE (c,fn,v) iolist
```

c, fn, and v are the same as for ENCODE.

iolist is the list to receive variables from the location specified by v. iolist conforms to the syntax of an input/output list.

```
    5 7
      PROGRAM ADD (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      DIMENSION CARD (8), INK (77)
    2 READ (5,100) KEY1,CARD
  100 FORMAT (I1,7A10,A9)
      IF (EOF(5)) 80,90
   90 IF (KEY1-2) 3,8,3
    3 CALL ERROR1
      GO TO 2
    8 WRITE (6,300) CARD
  300 FORMAT (1H1,7A10,A7///)
      DECODE (77,17,CARD) INK
   17 FORMAT (77I1)
      ITOT = 0
      DO 4 I = 1,77
    4 ITOT = ITOT + INK(I)
      ISAVE = ITOT
      WRITE (6,200) ISAVE
  200 FORMAT (19X,*TOTAL OF 77 SCORES ON CARD = *,I10)
   80 STOP
      END
      SUBROUTINE ERROR1
      WRITE (6,1)
    1 FORMAT (5X,*NUMBER IS NOT 2*)
      RETURN
      END
```

(An explanation of this program appears in part 2).

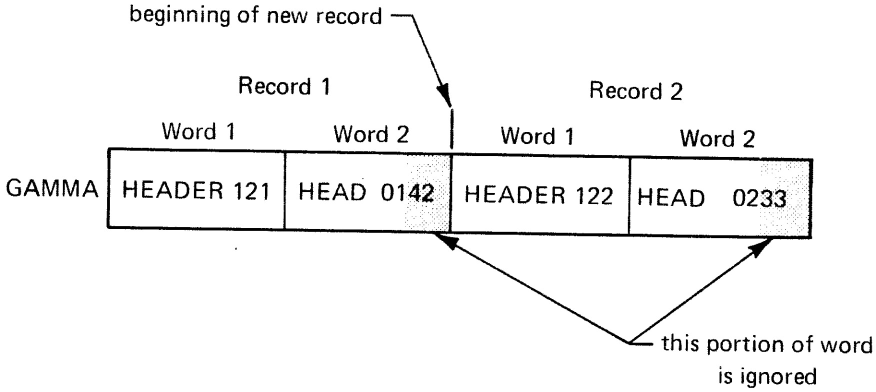
DECODE is a core-to-core transfer of data similar to READ. Only data in central memory or small core can be transferred by the DECODE statement. Display code characters in a variable or an array, v, are converted under format specifications and stored in the list variables, iolist. DECODE reads from a string of display code characters in an array or variable in central memory or small core; whereas the READ statement reads from an input device. Both statements convert data according to the format specification, fn. Using DECODE, however, the same information can be read several times with different DECODE and FORMAT statements.

Starting at the named location, v, data is transmitted according to the specified format and stored in the list variables. If the number of characters per record is not a multiple of 10 (a display code word contains 10 display code characters) the balance of the word is ignored. However, if the number of characters specified by the list and the format specification exceeds the number of characters per record, an execution error message is printed. If DECODE attempts to process an illegal display code character, or a character illegal under a given specification, that character is treated as a blank and conversion continues. Under the SCOPE operating system, a binary zero byte in the low order bits of a word is used to terminate a unit record. If DECODE encounters a zero character (6 bits of binary zeros), other than at the end of a line, the character is interpreted as a colon and conversion continues.

Example:

$c \neq$ multiple of 10

```
   DECODE (16,1,GAMMA) X,B,C,D
 1 FORMAT (2A8)
```



Data transmitted under this DECODE specification would appear in storage as follows:

```
X=HEADER 1
B=21HEAD
C=HEADER 1
D=22HEAD
```

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

```
LOC(1)SSSSSxxxxx
.
.
.
LOC(6)xxxxxDDDDD
```

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 display code 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.

Using the R specification, the example above could be shortened to:

```
  ENCODE(10,1,NAME)LOC(1),LOC(6)
1 FORMAT(A5,R5)
```

This chapter covers input/output lists and FORMAT statements. Input/output statements, which include READ and WRITE, are covered in section 9.

INPUT/OUTPUT LISTS

The list portion of an input/output statement specifies the items to be read or written and the order of transmission. The input/output list can contain any number of elements. List items are read or written sequentially from left to right. Their order must correspond to the FORMAT specification associated with the list, and they must be separated by commas.

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 list consists of a variable name, an array name, an array element name, or an implied DO list. On output the data list can include Hollerith constants and arithmetic expressions.

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

An array name without subscripts in an input/output list specifies the entire array in the order in which it is stored. The entire array (not just the first word of the array) is read or written.

Subscripts in an input/output list may be any valid subscript (section 2).

Examples:

```
READ 100,A,B,C,D
READ 200,A,B,C(I),D(3,4),E(I,J,7),H
READ 101,J,A(J),I,B(I,J)
READ 202,DELTA
READ 102, DELTA(5*J+2,5*I-3,5*K),C,D(I+7)
READ 3,A,(B,C,D),(X,Y)
```

An implied DO list is a list followed by a comma and an implied DO specification, all enclosed in parentheses.

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

$i = m_1,m_2,m_3$ $\qquad\qquad$ $i = m_1,m_2$

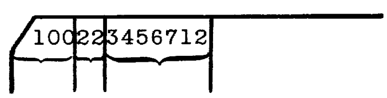
The elements i, m_1, m_2, and m_3 have the same meaning as in the DO statement. The range of a DO-implied specification is that of the DO-implied list. The values of i, m_1, m_2, and m_3 must not be changed within the range of the DO implied list by a READ statement.

On input or output, the list is scanned and each variable in the list is paired with the field specification provided by the FORMAT statement. After one item has been input or output, the next format specification is taken together with the next element of the list, and so on until the end of the list.

Example:

```
   READ (5,20)L,M,N
20 FORMAT (I3,I2,I7)
```

Input record



100 is read into the variable L under the specification I3, 22 is read into M under the specification I2, and 3456712 is read into N under specification I7.

ARRAY TRANSMISSION

Input/output of array elements may be accomplished by using an implied DO loop. The list of variables followed by the DO loop index, is enclosed in parentheses to form a single element of the input/output list

Example:

```
READ (5,100) (A(I),I=1,3)
```

has the same effect as the statement

```
READ (5,100) A(1),A(2),A(3)
```

The general form for an implied DO loop is:

$$(\ldots((\mathrm{list},i_1=m_1,m_2,m_3),i_2=j_1,j_2,j_3),\ldots,i_n=k_1,k_2,k_3)$$

m,j,k are unsigned integer constants or predefined positive integer variables. If m_3, j_3 or k_3 is omitted, a one is used for incrementing.

$i_1 \ldots i_n$ are index variables. An index variable should not be used twice in the same implied DO nest, but array names, array elements and variables may appear more than once.

The first index variable (i_1) defined in the list is incremented first. i_1 is set equal to m_1 and the associated list is transmitted; then i_1 is incremented by m_3, until m_2 is exceeded. When the first index variable reaches m_2, it is reset to m_1; the next index variable at the right (i_2) is incremented; and the process is repeated until the last index variable (i_n) has been incremented, until k_2 is exceeded.

The general form for an array is:

$(((A(I,J,K),i_1=m_1,m_2,m_3),i_2=n_1,n_2, n_3),i_3=k_1,k_2,k_3)$

Example:

```
READ 100,((A(JV,JX),JV=2,20,2),JX=1,30)
READ 200,(BETA(3*JON+7),JON=JONA,JONB,JONC)
READ 300,(((ITMLIST(I,J+1,K- 2),I=1,25),J=2,N),K=IVAR,IVMAX,4)
```

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

Examples of simple implied DO loop list items:

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

The following DO loop would have the same effect:

```
    DO 5 I=1,10
  5 READ 400, A(I)
```

Example:

CAT,DOG, and RAT will be transmitted 10 times each with the following iolist

```
(CAT, DOG, RAT, I=1,10)
```

Implied DO loops may be nested.

Example:

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

Equivalent to the following:

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

The list is similar to the nest of DO loops:

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

Example:

The following list item transmits nine elements into the array E in the order: E(1,1), E(1,2), E(1,3), E(2,1), E(2,2), E(2,3), E(3,1), E(3,2), E(3,3)

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

Example:

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

Data is transmitted in the following sequence:

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

Data can be read from or written into part of an array by using the implied DO loop.

Examples:

```
      READ (5,100)  (MATRIX(I),I=1,10)
  100 FORMAT (F7.2)
```

Data (consisting of one constant per record) is read into the first 10 elements of the array MATRIX. The following statements would have the same effect:

```
      DO 40 I = 1,10
   40 READ (5,100) MATRIX(I)
  100 FORMAT (F7.2)
```

In this example, the statements are equivalent; numbers are read, one from each card, into the elements MATRIX(1) through MATRIX(10) of the array MATRIX. In the second case, however, the READ statement is encountered each time the DO loop is executed; and a new card is read for each element of the array. Each execution of a READ statement reads at least one record regardless of the FORMAT statement.

```
    READ (5,100) (MATRIX(I),I=1,10)
100 FORMAT (F7.2)
```

In the above statements, the implied DO statement is part of the READ statement; therefore, the FORMAT statement specifies the format of the data input and determines when a new card will be read.

If statement 100 FORMAT (F7.2) had been 100 FORMAT (4F20.10), only three cards would be read.

To read data into an entire array, it is necessary only to name the array in a list without any subscripts.

Example:

```
DIMENSION B (10,15)
READ 13,B
```

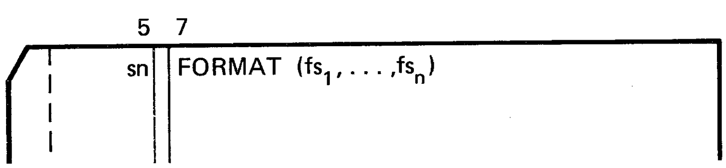
is equivalent to

```
READ 13 ((B(I,J),I=1,10),J=1,15)
```

The entire array B will be transmitted in both cases.

FORMAT STATEMENT

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. A FORMAT statement is required to transmit formatted information.



sn	Statement label which must appear
$fs_1,\ldots,fs_n$	Set of one or more field specifications separated by commas and/or slashes and optionally grouped by parentheses

Example:

```
    READ (5,100) INK,NAME,AREA
100 FORMAT  (10X,I4,I2,F7.2)
```

FORMAT is a non-executable statement which specifies the format of data to be moved between input/output device and central memory or small core. It is used in conjunction with read and write statements, and it may appear anywhere in the program.

The FORMAT specification is enclosed in parentheses. Blanks are not significant except in Hollerith field specifications.

Each item in an input/output list is associated with a corresponding field specification in a FORMAT statement. The FORMAT statement specifies the external format of the data, and the type of conversion to be used, and defines the length of the FORTRAN record or records.

The type of conversion should correspond to the type of the variable in the input/output list. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete.

For example:

```
      INTEGER N
      READ (5,100) N
  100 FORMAT (F10.2)
```

A floating point number is assigned to the variable N which could cause unpredictable results if N is referenced later as an integer.

DATA CONVERSION

The following types of data conversion are available:

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

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

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

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

FIELD SEPARATORS

Field separators are used to separate specifications and groups of specifications. The format field separators are the slash (/) and the comma. The slash is also used to specify demarcation of formatted records.

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, 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.

Complex data items are converted on input/output as two independent floating point quantities. The format specification uses two single precision conversion elements.

Example:

```
   COMPLEX A,B,C,D
   PRINT 10,A
10 FORMAT (F7.2,E8.2)
   READ 11,B,C,D
11 FORMAT (2E10.3,2(F8.3,F4.1))
```

Data of differing types may be read by the same FORMAT statement. For example:

```
10 FORMAT (I5,F15.2)
```

specifies two numbers, the first of type integer, the second of type real.

```
   READ (5,15) NO,NONE,INK,A,B,R
15 FORMAT (3I5,2F7.2,A4)
```

reads 3 integer variables

reads 2 real variables

reads 1 alphanumeric variable

Iw INPUT

The I conversion is used to input decimal integer constants.

Iw

w is a decimal integer constant designating the total number of characters in the field including signs and blanks.

The plus sign may be omitted for positive integers. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. An all blank field is considered to be minus zero. Decimal points are not permitted. The value is stored in the specified variable. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will terminate execution.

Example:

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

Input Card:

```
139|bb-15bb|18|bb7|bb|b1b4
 3     7    2   3   2   4
```

In storage:

I contains 139
J contains -1500
K contains 18
L contains 7
M contains -0
N contains 104

Iw OUTPUT

The I specification is used to output decimal integer values.

Iw

w is a decimal integer constant designating the total number of characters in the field including signs and blanks. If the integer is positive the plus sign is suppressed. Only numbers in the range of $-2^{48}+1$ to $2^{48}-1$ are output correctly; even though, internally, integers outside this range can be generated. If the absolute value of the integer is greater than $2^{48}-1$ ($2^{48}-1 = 281\ 474\ 976\ 710\ 655$), an X is printed in the field.

The specification Iw outputs a number in the following format:

ba...a

b	Minus sign if the number is negative, or blank if the number is positive
a...a	May be a maximum of 15 digits

The output quantity is right justified with blanks on the left. If the field is too short, characters are stored from the right, and an asterisk occupies the leftmost position.

Example:

```
     PRINT 10,I,J,K

  10 FORMAT (I9,I10,I5)
```

I contains -3762
J contains +4762937
K contains +13

Result:

```
bbb-3762bbb4762937bbb13
```

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

1st blank taken as printer control character

Example:

```
     WRITE (6,100)N,M,I

 100 FORMAT (I5,I6,I9)
```

N contains +20
M contains -731450
I contains +205

Result:

```
bb20*31450bbbbbb205
```

$\underbrace{\texttt{bb20}}_{4}\underbrace{\texttt{*31450}}_{6}\underbrace{\texttt{bbbbbb205}}_{9}$

1st blank taken as printer control character

specification too small * indicates most significant digits missing

Ew.d OUTPUT

E specifies conversion between an internal real value and an external number written with exponent.

Ew.d

w is an unsigned integer designating the total number of characters in the field. w must be wide enough to contain digits, plus or minus signs, decimal point, E, the exponent and blanks. Generally, $w \geq d+6$ for negative numbers and $w \geq d+5$ for positive numbers. Positive numbers need not reserve a space for the sign of the number. If the field is not wide enough to contain the output value, an asterisk is inserted in the leftmost position of the field. If the field is longer than the output value, the quantity is right justified with blanks on 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.

d specifies the number of digits to the right of the decimal within the field. If d is zero or blank, the decimal point and digits to the right of the decimal do not appear.

The specification Ew.d produces output in the following format:

b.a...a±eee or	-308 ≤ eee ≤ 337
b.a...aE±ee	0 ≤ ee ≤ 99
b	Minus sign if the number is negative, or blank if the number is positive
a...a	Most significant digits
eee	Digits in the exponent

Examples:

```
   PRINT 10,A
10 FORMAT (E10.3)
```

A contains -67.32 or +67.32

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

```
   PRINT 10,A
10 FORMAT (E13.3)
```

Result: `bbb-.673E+02` or `bbbb.673E+02`

```
   PRINT 10,A
10 FORMAT (1H ,E8.3)
```

A contains -67.32
no provision for - sign

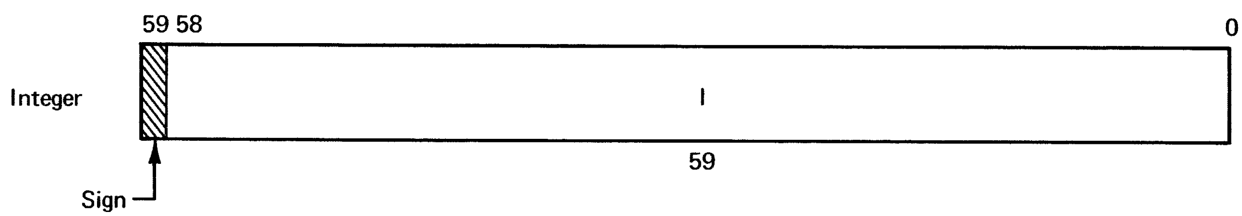
Result: `*.67E+02`

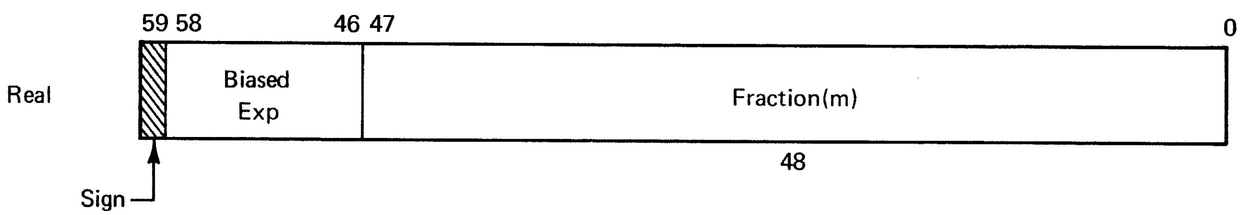
```
   PRINT 10,A
10 FORMAT (1H ,E10.6)
```

1H is the carriage control character

Result: `*.6732E+02`

If an integer variable is output under the Ew.d specification, results are unpredictable since the internal format of real and integer values differ. An integer value does not have an exponent and will be printed, therefore, as a very small value or 0.0.





Ew.d INPUT

E specifies conversion between an external number written with an exponent and an internal real value.

Ew.d

w is an unsigned integer designating the total number of characters in the field, including plus or minus signs, digits, decimal point, E and exponent. If an external decimal point is not provided, d acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is:

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

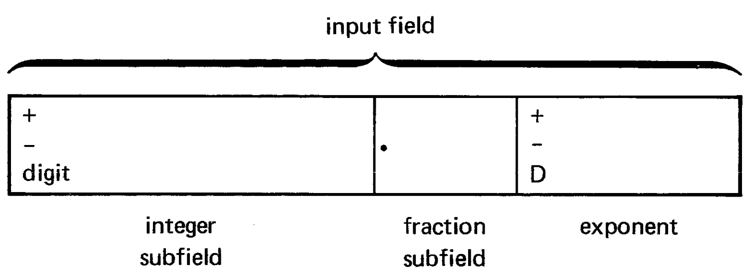
For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as: $3267 \times 10^{-8} \times 10^{5} = 3.267$.

If an external decimal point is provided, it overrides d. If d does not appear it is assumed to be zero.

In the input data, leading blanks are not significant; other blanks are interpreted as zeros.

An input field consisting entirely of blanks is interpreted as minus zero.

The following diagram illustrates the structure of the input field:



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

The fraction subfield begins with a decimal point and terminates with an E, +, - or the end of the input field. It may contain a string of digits.

The exponent subfield may begin with E, + or -. When it begins with E, the + is optional between E and the string of digits in the subfield.

For example, the following are valid equivalent forms for the exponent 3:

E+03, E 03,E03,E+3,E3, +3,+ 3,E + 3

The value of the string of digits in the exponent subfield must be less than 323.

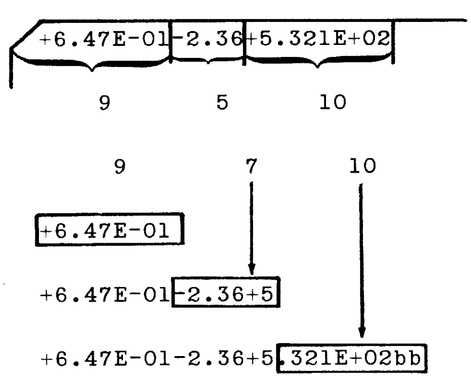
Valid subfield combinations:

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

If the field length specified by w in Ew.d is not the same as the length of the field containing the input number, incorrect numbers may be read, converted, and stored. The following example illustrates a situation where numbers are read incorrectly, converted and stored; yet there is no immediate indication that an error has occurred:

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

On the card, input quantities are in three adjacent fields, columns 1-24:



First, +647E-01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number -2.36+5 is read instead of -2.36. 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 number. Since the second specification incorrectly took two digits from the third number, the specification for the third number is now incorrect. The number .321E+02bb is read. Trailing blanks are treated as zeros; therefore the number .321E+0200 is read converted and placed in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Examples of Ew.d input specifications:

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	.14326	All subfields present
-12.437629E+1	E13.6	-124.37629	All subfields present
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 left of decimal contains only a minus sign and a plus sign appears instead of E in input field
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only
blanks	Ew.d	-0.	All subfields empty
1E1	E3.0	10.	No fraction subfield; input number converted as 1.x10
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
1.0E16	E6.3	10000000000.	

Fw.d OUTPUT

The F specification outputs a real number without a decimal exponent.

Fw.d

w is an unsigned integer which designates the total number of characters in the field including the sign (if negative) and decimal point. N must be $\geq$ d + 2.

d specifies the number of places to the right of the decimal point. If d is zero or omitted, the decimal point and digits to the right do not appear.

If the number is positive, the plus sign is suppressed. If the field is too short, one asterisk appears in the leftmost position of the output field. If the field is longer than required, the number is right justified with blanks on 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.

The specification Fw.d outputs a number in the following format:

```
  ┌────── decimal point
b...a.a...a
```

b Minus sign if the number is negative, or blank if the number is positive.

Examples:

Value of A	FORMAT Statement	PRINT Statement	Printed Result
+32.694	10 FORMAT (1H ,F6.3)	PRINT 10,A	32.694
+32.694	11 FORMAT (1H ,F10.3)	PRINT 11,A	bbbb32.694
-32.694	12 FORMAT (1H ,F6.3)	PRINT 12,A	*2.694 (no provision for minus sign and most significant digit)
.32694	13 FORMAT (1H ,F4.3,F6.3)	PRINT 13,A,A	.327bb.327

The specification 1H is the carriage control character.

Fw.d INPUT

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

Examples of the F format specification:

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field
-4.7366	F7	-4.7366	No d in specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.2	.62543	Decimal point overrides d of specification
+144.15E-03	F11.2	.14415	Exponents are allowed in F input, and may have P scaling
5bbbb	F5.2	500.00	No fraction subfield; input number converted as 50000×10^{-2}
bbbbb	F5.2	-0.00	Blanks in input field interpreted as -0

Gw.d INPUT

Input under control of G specification is the same as for the E specification. The rules which apply to the E specification apply to the G specification.

Gw.d

w — Unsigned integer which designates the total number of characters in the field including E, digits, sign, and decimal point

d — Number of places to the right of the decimal point

Example:

```
   READ (5,11) A,B,C
11 FORMAT (G13.6,2G12.4)
```

Gw.d OUTPUT

Output under control of the G specification is dependent on the size of the floating point number being converted. The number is output under the F conversion unless the magnitude of the data exceeds the range which permits effective use of the F. In this case, it is output under E conversion with an exponent.

Gw.d

w — Unsigned integer which designates the total number of characters in the field including digits, signs and decimal point, the exponent E, and any leading blanks.

d — Number of significant digits output.

If a number is output under the G specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field E±00). Therefore, for output under G conversion w must be greater than or equal to d + 6. The six extra spaces are required for sign and decimal point plus four spaces for the exponent field.

Example:

```
    PRINT 200,YES          YES contains 77.132
200 FORMAT (G10.3)
```

Output: `b77.1bbbb`   b denotes a blank

If the decimal point is not within the first d significant digits of the number, the exponential form is used (G is treated as if it were E).

Example:

```
     PRINT 100, EXIT
 100 FORMAT (G10.3)
```

EXIT contains 1214635.1

Output: `.1215E+07`

Example:

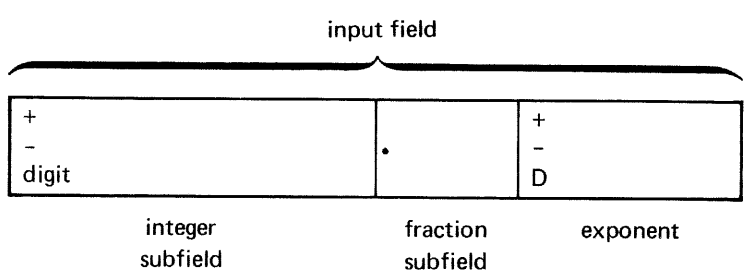
```
     READ (5,50) SAMPLE
     .
     .
     .
     WRITE (6,20) SAMPLE
  20 FORMAT (1X,G17.8)
```

Data read by READ statement	Data Output	Format Option
.1415926535bE-10	.141592653E-10	E conversion
.8979323846	.89793238	F conversion
2643383279.	.264338328E+10	E conversion
-693.9937510	-693.99375	F conversion

Dw.d OUTPUT

Dw.d

Type D conversion is used to output double precision variables. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. 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 of type D output:

```
      DOUBLE A,B,C
      A = 111111.11111
      B = 222222.22222
      C = A + B
      PRINT 10,A,B,C
   10 FORMAT (3D23.10)

      .11111111111D+06     .22222222222D+06     .33333333333+06
```

The specification Dw.d produces output in the following format:

decimal point

b.a...a±eee	-308 ≤ eee ≤ 337
b.a...aD±ee	0 ≤ ee ≤ 99

b	Minus sign if the number is negative, or blank if the number is positive
a...a	Most significant digits
ee	Digits in the exponent

Dw.d INPUT

D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield.

Ow INPUT

Octal values are converted under the O specification.

Ow

w is an unsigned integer designating the total number of characters in the field. The input field may contain a maximum of 20 octal digits. Blanks are allowed and a plus or minus sign may precede the first octal digit. Blanks are interpreted as zeros and an all blank field is interpreted as minus zero. A decimal point is not allowed.

The list item corresponding to the Ow specification should be integer.

Example:

```
      INTEGER P,Q,R
      READ 10,P,Q,R
   10 FORMAT (010,012,02)
```

Input Card:

```
 3737373737 666b6644b444 -0
     10          12       2
```

Input storage (octal representation):

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

Ow OUTPUT

The O specification is used to output octal integer values, the internal representation of the number.

Ow

w is an unsigned integer designating the total number of characters in the field. If w is less than 20, the rightmost digits are output. For example, if the contents of location P were output with the following statement the digit 3737 would be output.

```
      WRITE (6,1) P                location P 00000000003737373737
  100 FORMAT (1X,04)
```

If w is greater than 20, the 20 octal digits (20 octal digits = a 60- bit word) are right justified with blanks on the left.

For example, if the contents of location P are output with the following statement

```
      WRITE (6,200) P
  200 FORMAT (1X,022)
```

Output would appear as follows:

```
bb00000000003737373737            b = blank
```

A negative number is output in one's complement internal form.

Example:

```
I = -11
WRITE (6,200) I
```

Output would appear as follows:

```
bb77777777777777777764
```

The specification Ow produces output in the following format:

a...a

a...a   octal digits

Aw INPUT

The A specification is used to input alphanumeric characters.

Aw

w is an unsigned integer designating the total number of characters in the field.

Alphanumeric information is stored as 6-bit display code characters, 10 characters per 60-bit word. For example, the digit 4 when read under A specification is stored as a display code 37. If w is less than 10, the input quantity is stored left justified in the word; the remainder of the word is filled with blanks.

Example:

```
    READ (5,100) A
100 FORMAT (A7)
```

Input record:

```
EXAMPLE
```

When EXAMPLE is read it is stored left justified in the 10 character word

```
1234567890
EXAMPLE
```

If w is greater than 10, the rightmost 10 characters are stored and remaining characters are ignored.

Example:

```
    READ (5,200)B
200 FORMAT (A13)
```

Input record:

```
 1           13
/SPECIFICATION
```

In storage:

```
12345678910
CIFICATION
```

```
      READ (5,10) Q,P,R
   10 FORMAT (A10,A10,A5)
```

Input record:

```
/THIS IS AN EXAMPLE I KNOW
    10         10       5
```

In storage:

```
 12345678910
Q THIS IS AN
P  EXAMPLE I
R  KNOW
```

Aw OUTPUT

The A specification is used to output alphanumeric characters.

Aw

w is an unsigned integer designating the total number of characters in the field. If w is less than 10, the leftmost characters in the word are printed. For example, if the contents of location A in the Aw input example are output with the following statements:

```
      WRITE (6,300)A
  300 FORMAT (1X,A4)
```

In storage:

```
A  EXAMPLE
```

Characters EXAM are output

If w is greater than 10, the value is right justified in the output field with blanks on the left. For example, if A in the previous example is output with the following statements:

```
      WRITE (6,400)A
  400 FORMAT (1X,A12)
```

Printed output appears as follows:

```
bbEXAMPLEbb
```
b = blank

Rw INPUT

w is an unsigned integer designating the total number of characters in the field. The R specification is the same as the A specification with the following exception. If w is less than 10, the input characters are stored right justified, with binary zero fill on the left.

Example:

```
      READ (5,600) HOO,RAY
  600 FORMAT (R10,R5)
```

Input card:

$\underbrace{\texttt{RESULTS OF}}_{10}\underbrace{\texttt{ TEST}}_{5}$

In storage:

```
HOO  RESULTSbOF
RAY  0...00bTEST
```
b = blank

Rw OUTPUT

Rw

w is an unsigned integer designating the total number of characters in the field.

This specification is the same as the A specification with the following exception. If w is less than 10, the rightmost characters are output. For example, if RAY from the previous example is output with the following statements:

```
      WRITE (6,700) RAY
  700 FORMAT (1X,R3)
```
Characters EST are output.

Lw INPUT

The L specification is used to input logical variables.

Lw

w is an unsigned integer designating the total number of characters in the field.

If the first non-blank character in the field is T, the logical value .TRUE. is stored in the corresponding list item which should be of type logical. If the first non-blank character is F, the value .FALSE. is stored. If the first non-blank character is not T or F, a diagnostic is printed. An all blank field has the value .FALSE.

Lw OUTPUT

Lw

w is an unsigned integer designating the total number of characters in the field.

Variables output under the L specification should be of type logical. A value of .TRUE. or .FALSE. in storage is output as a right justified T or F with blanks on the left.

Example:

```
      LOGICAL I,J,K              I contains -0
      PRINT 5,I,J,K              J contains 0
    5 FORMAT (3L3)               K contains -0
```

Output:

```
bTbbFbbT
```

SCALE FACTORS

The scale factor P is used to change the position of a decimal point of a real number when it is input or output. Scale factors may precede D, E, F and G format specifications.

nPDw.d

nPEw.d

nPFw.d

nPGw.d.

nP

n is the scale factor. It is a positive (unsigned) or negative (signed) integer constant. w is an unsigned integer constant designating the total width of the field. d determines the number of digits to the right of the decimal point.

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

Once a scale factor is specified, it holds for all D, E, F, and G specifications in that FORMAT statement until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G specifications, a zero scale factor, 0P must precede a specification.

Example:

```
15 FORMAT(2PE14.3,F10.2,G16.2,0P4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specification. The 0P scale factor restores normal scaling ($10^0 = 1$) for the subsequent specification 4F13.2.

A scaling factor may appear independently of a D, E, F or G specification. It holds for all subsequent D, E, F or G specifications within the same FORMAT statement,until changed by another scaling factor.

Example:

```
FORMAT(3P,5X,E12.6,F10.3,0PD18.7,-1P,F5.2)
```

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

The specification (3P,3I9,F10.2) is the same as the specification (3P3I9,3PF10.2).

Fw.d SCALING

INPUT

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

OUTPUT

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

Specification	Output Representation
F13.6	3.141593
1PF13.6	31.415927
3PF13.6	3141.592654
-1PF13.6	.314159
-3PF13.6	.003142

Ew.d SCALING

Ew.d scaling on input is the same as Fw.d scaling on input.

OUTPUT

The scale factor has the effect of shifting the output number left n places while reducing the exponent by n. Using 3.1415926536, the following are output representations corresponding to scaled E specifications:

Specification	Output Representation
E20.2	3.14 E+00
1PE20.2	31.42 E-01
2PE20.2	314.16 E-02
3PE20.2	3141.59 E-03
4PE20.2	31415.93 E-04
5PE20.2	314159.27 E-05
-1PE20.2	0.31 E+01

Gw.d SCALING

INPUT

Gw.d scaling on input is the same as Fw.d scaling on input.

OUTPUT

The effect of the scale factor is suspended unless the magnitude of the data to be converted is outside the range that permits the effective use of F conversion.

X

The X specification is used to skip characters in an input line and insert blank characters in an output line. It is not associated with a variable in the input/output list.

nX

n   Number of characters to be skipped or the number of blanks to be inserted on output, n blanks are inserted in an output record.

0X is ignored. bX is interpreted as 1X. The comma following X in the specification list is optional.

Example:

```
    WRITE (6,100) A,B,C                A = -342.743
100 FORMAT (F9.4,4X,F7.5,4X,I3)        B = 1.53190
                                       C = 22
```

Output record:

```
    -342.743bbbb1.53190bbbbb22
```

b is a blank

on input n columns are skipped.

Example:

```
   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

Example:

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

Result: `7bbbbbbb13.60bbbbbbb.146237E+04`

nH OUTPUT

The H specification is used to output strings of alphanumeric characters and like X, H is not associated with a variable in the input/output list.

nH

n	Number of characters in the string including blanks. n may not exceed 136 characters.
H	Denotes a Hollerith field. The comma following the H specification is optional.

For example, the statement:

```
   WRITE (6,1)
 1 FORMAT (15HbENDbOFbPROGRAM)
```

can be used to output the following on the output listing.

```
   END OF PROGRAM
```

Examples:

Source program:

```
   PRINT 20
20 FORMAT (28HbBLANKSbCOUNTbINbANbHbFIELD.)
```

produces output record:

```
   BLANKSbCOUNTbINbANbHbFIELD.
```

Source program:

```
   PRINT 30,A                          A contains 1.5
30 FORMAT (6HbLMAX=,F5.2)
```

produces output record:

```
   LMAX=b1.50
```

nH INPUT

The H specification can be used to read Hollerith characters into an existing H field within the FORMAT statement.

Example:

Source program:

```
   READ 10
10 FORMAT (27Hbbbbbbbbbbbbbbbbbbbbbbbbbbb)
```

Input card:

```
bTHIS IS A VARIABLE HEADING
```

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:

```
THIS IS A VARIABLE HEADING
```

... ≠...≠

Character strings delimited by a pair of * or ≠ symbols can be used as alternate forms of the H specification, and can appear anywhere the H form of the Hollerith constant can appear. The paired symbols delineate the Hollerith field. This specification need not be separated from other specifications by commas. If the Hollerith field is empty, or invalidly delimited a fatal execution error occurs, and an error message is printed.

An asterisk cannot be output using the specification * *. For example,

```
  PRINT 1
1 FORMAT (*ABC*DE*)
```

The second * in the FORMAT statement causes the specification to be interpreted as *ABC* and DE*, which is not valid.

The H specification or ≠...≠ could be used to output this correctly:

```
  PRINT 1
1 FORMAT (7H ABC*DE)
```

Output appears as follows: `ABC*DE`

```
  PRINT 2
2 FORMAT (≠ ABC*DE≠)
```

Output appears as follows: `ABC*DE`

≠ can be represented within ≠...≠ by two consecutive ≠ symbols.

Example:

```
  PRINT 3
3 FORMAT (≠ DON≠≠T≠)
```

Output examples:

```
      PRINT 10
   10 FORMAT (* SUBTOTALS*)
```

produces the following output:

```
      SUBTOTALS
```

```
      WRITE (6,20)
   20 FORMAT (≠bRESULT OF CALCULATIONS IS AS FOLLOWS≠)
```

produces the following output:

```
      RESULT OF CALCULATIONS IS AS FOLLOWS
```

```
      PRINT 1, ≠SQRT≠, SQRT(4.)
    1 FORMAT (A10,E10.2)
```

produces the following output:

```
      SQRT       2.0
```

Note: ≠ is output as ' on some printers.

The *...* specification can be used to read alphanumeric characters into an existing *...* field within the FORMAT statement. Characters are stored in the heading until either an asterisk is encountered in the input field or all the spaces in the format specification are filled. For example, if the format specification contains 10 spaces and the 6th character in the input field is an asterisk, all the characters to the left of the asterisk are stored in the heading; and the remaining character positions in the heading are filled with blanks.

Input examples:

Source program:

```
      READ 10
   10 FORMAT (*bbbbbbbbbbbbbbbbbbbbb*)
```

Input card:

```
bFORTRAN FOR THE 6600
```

A subsequent reference to statement 10 in an output control statement:

PRINT 10 produces:

```
      FORTRAN FOR THE 6600
```

Source program:

```
     READ 10
  10 FORMAT (*bbbbbbbb*)
```

Input card:

```
bHEAD*
```

PRINT 10 produces:

```
HEADbbb
```

```
     READ (5,600)
 600 FORMAT (*bbbbbbbbbbbb*)
```

b is a blank

Input record:

```
MAGNITUDE CALCULATION
```

WRITE (6,600) produces:

```
MAGNITUDE C
```

FORTRAN RECORD /

The slash indicates the end of a FORTRAN record anywhere in the FORMAT specification list. Where a / is used a comma is not required, but it is allowed, to separate field specification elements. Consecutive slashes may appear in a list and need not be separated from other list elements by commas. During output, the slash indicates the end of a record. During input, it specifies further data comes from the next record.

Example:

```
     PRINT 10
  10 FORMAT (6X, 7HHEADING///3X, 5HINPUT, 8H OUTPUT)
```

Printout:

```
      HEADING ________________ line 1
              ______ (blank)___ line 2
              ______ (blank)___ line 3
INPUT OUTPUT ________________ line 4
```

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

A repetition factor can be used to indicate multiple slashes.

n(/)

n — Unsigned integer indicating the number of slashes required. n - 1 lines are skipped on output.

Example:

```
      PRINT 15, (A(I),I=1,9)
   15 FORMAT (8HbRESULTS4(/),(3F8.2))
```

Format statement 15 is equivalent to:

```
   15 FORMAT (8HbRESULTS//// (3F8.2))
```

Printout:

```
RESULTS ________________________________________ line 1
         ________________________ (blank) ______ line 2
         ________________________ (blank) ______ line 3
         ________________________ (blank) ______ line 4
  3.62   -4.03   -9.78 _________________________ line 5
 -6.33    7.12    3.49 _________________________ line 6
  6.21   -6.74   -1.18 _________________________ line 7
```

Example:

```
      DIMENSION B(3)
      READ (5,100)IA,B
  100 FORMAT (I5/3E7.2)
```

These statements read two records, the first containing an integer number, and the second containing three real numbers.

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

In storage:

A -11.6
B .325
C 46.327
D -14.261

Printout:

```
b-.12E+02bbb.33E+00
46.327-14.261
```

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

Printout:

```
b-.12E+02bbb.33E+00 ————————————— line 1
———————————————— (blank) ——— line 2
46.327-14.261 ——————————————————— line 3
```

REPEATED FORMAT SPECIFICATION

FORMAT specifications may be repeated by preceding the control characters D,E,F,G,I,A,R,L or O by an unsigned integer giving the number of repetitions required.

`100 FORMAT (3I4,2E7.3)` is equivalent to: `100 FORMAT (I4,I4,I4,E7.3,E7.3)`

`50 FORMAT (4G12.6)` is equivalent to: `50 FORMAT (G12.6,G12.6,G12.6,G12.6)`

A group of specifications may be repeated by enclosing the group in parentheses and preceding it with the repetition factor.

```
1 FORMAT (I3,2(E15.3,F6.1,2I4))
```

is equivalent to the following specification if the number of items in the input/output list do not exceed the format conversion codes:

```
1 FORMAT (I3,E15.3,F6.1,I4,I4,E15.3,F6.1,I4,I4)
```

Two levels of parentheses, in addition to the parentheses required by the FORMAT statement, are the maximum allowed when groups of specifications are repeated. The following example illustrates maximum nesting of parentheses:

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

If the number of items in the input/output list are fewer than the number of format codes in the FORMAT statement, excess FORMAT codes are ignored.

If the number of items in the input/output list exceed the format conversion codes, when the final right parenthesis in the FORMAT statement is reached, the line formed internally is output. The FORMAT control then scans to the left looking for a right parenthesis within the FORMAT statement. If none is found the scan stops when it reaches the beginning of the FORMAT specification. If, however, a right parenthesis is found, the scan continues to the left until it reaches the field separator which precedes the left parenthesis pairing this right parenthesis. Output resumes with the FORMAT control moving right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered.

Example:

```
      READ (5,300)I,J,E,K,F,L,M,G,N,R
  300 FORMAT (I3,2(I4,F7.3),I7)
```

is equivalent to storing data in I with format I3, J with I4, E with F7.3, K with I4, F with F7.3, L with I7. Then a new record is read; data is stored in M with the format I4, G with F7.3, N with I4 and R with F7.3.

```
      READ (5,100) NEXT, DAY, KAT, WAY, NAT, RAY,  MAT
  100 FORMAT (I7,(F12.7,I3))
```

NEXT is input with format I7, DAY is input with F12.7, KAT is input with I3. The FORMAT statement is exhausted, (the right parenthesis has been reached) a new card is read, and the statement is rescanned from the group (F12.7,I3). WAY is input with the format F12.7, NAT with I3, RAY with F12.7, MAT with I3.

PRINTER CONTROL CHARACTER

The first character of a printer output record is used for carriage control and is not printed. It appears in all other forms of output as data.

The printer control characters are as follows:

Character	Action
Blank	Space vertically one line then print
0	Space vertically two lines then print
1	Eject to the first line of the next page before printing
+	No advance before printing; allows overprinting
Any other character	Refer to the SCOPE Reference Manual

For output directed to the card punch or any device other than the line printer, control characters are not required. If carriage control characters are transmitted to the card punch, they are punched in column one.

Carriage control characters can be generated by any means; however, the H specification is frequently used.

Example:

```
      FORMAT (1HO,F7.3,I2,G12.6)

      FORMAT (1H1,I5,*RESULT = *,F8.4)
```

The *...* specification can be used:

```
FORMAT (*1*,I4,2(F7.3))
```

The blank printer control character can be transmitted by the X specification.

Example:

```
FORMAT (1X,I4,G16.8)
```

Carriage control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash.

Example:

```
      PROGRAM LOGIC(INPUT,OUTPUT,TAPE5=INPUT)
      LOGICAL MALE,PHD,SINGLE,ACCEPT
      INTEGER AGE
      PRINT 20
   20 FORMAT (*1                 LIST OF ELIGIBLE CANDIDATES*)
    3 READ (5,1) LNAME,FNAME,MALE,PHD,SINGLE,AGE
    1 FORMAT (2A10,3L5,I2)
      IF (EOF(5))6,4
    4 ACCEPT = MALE .AND. PHD .AND. SINGLE .AND. (AGE .GT. 25 .AND.
     S    AGE .LT. 45)
      IF (ACCEPT) PRINT 2,LNAME,FNAME,AGE
    2 FORMAT (1H0,2A10,3X,I2)
      GO TO 3
 6    STOP
      END
```

output starts on new page

double spacing

```
                LIST OF ELIGIBLE CANDIDATES

 JOHN S.   SLIGHT        26

  JUSTIN   BROWN         30
```

Tn

This specification is a column selection control.

Tn

n Unsigned integer ≤ 136 for input/output and ≤ 150 for ENCODE/DECODE. If n=zero, column 1 is assumed.

When Tn is used, control skips columns until column n is reached, then the next format specification is processed. Using card input, if n > 80 the column pointer is moved to column n, but a succeeding specification would read only blanks.

```
   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.

```
   READ (20,40)
40 FORMAT (T10,*COLUMN1*)
```

The first nine characters of the input record are skipped and the next seven, COLUMN1, are read from input file 20.

With T specification, the order of a list need not be the same as the printed page or card input, and the same information can be read more than once.

Example:

```
     5 7
       PROGRAM TEST (OUTPUT)
     1 FORMAT (12(10H0123456789))
       PRINT 1
       PRINT 60
    60 FORMAT (T80,*COMMENTS*,T60,*HEADING4*,T40,
      X        *HEADING3*,T20,*HEADING2*,T2,*HEADING1*)
       PRINT 10
    10 FORMAT (20X*THIS IS THE END OF THIS RUN*T52*...HONEST*)
       PRINT 1
       STOP
       END
```

```
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
HEADING1           HEADING2            HEADING3            HEADING4            COMMENTS
                   THIS IS THE END OF THIS RUN    ...HONEST
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Since the first character in a line output to the printer is used for printer control, T2 is output in the first print position.

Example:

The following example shows that it is possible to destroy a previously formed field inadvertently. The specification T5 destroys part of the Hollerith specification 10H DISASTERS.

```
1 FORMAT (10H DISASTERS,T5,3H123)
  PRINT 1
```

produces the following output:

```
DIS123ERS
```

EXECUTION TIME FORMAT STATEMENTS

Variable FORMAT statements can be read in as part of the data at execution time and used by READ, WRITE, statements later in the program. The format is read in as alphanumeric text under the A specification and stored in an array or a simple variable, or it may be included in a DATA statement. The format must consist of a list of format specifications enclosed in parentheses, but without the word FORMAT or the statement label.

For example, a data card could consist of the characters:

```
(E7.2,G20.5,F7.4,I3)
```

The name of the array containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The array name, which appears with or without subscripts, specifies the location or the first word of the FORMAT information.

For example, assume the following FORMAT specifications:

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

This information on an input card can be read by the statements of the program such as:

```
      DIMENSION IVAR(3)
      READ 1, IVAR
     1FORMAT (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
```

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)
```

The FORTRAN Extended compiler is called from the library and executed by an FTN control card. The FTN control card calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This control card may be in any of the following forms:

```
1
FTN (p1,p2,...,pn) comments

1
FTN. comments

1
FTN, p1,...,pn. comments
```

Example:

```
FTN (A,LRN,G,S=0)
```

The optional parameters, $p_1,...,p_n$ may be in any order within the parentheses. All parameters, with the exception of the list control options, must be separated by commas. If no parameters are specified, FTN is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the control statement syntax for job control statements as defined in the SCOPE Reference Manual. Card columns following the right parenthesis, or period, can be used for comments; they are ignored by the compiler, but are printed on the dayfile.

$(p_1,...,p_n)$

Default options are used for omitted parameters. Default options are set when the system is installed; but since installations can change default values, the user should consult his installation for possible changes.

In the following description of the FTN control card parameters: lfn is a 1-7 alphanumeric file name. The first letter must be alphabetic.

The first improperly formed parameter terminates the FTN control card scan. When an error is detected, a dayfile entry is made. A ten character segment of the control card is printed with an asterisk positioned beneath the approximate column in which the error occurred.

```
* POINTS TO FTN CONTROL CARD ERROR
```

Example:

Dayfile

```
22.14.11 FTN(I=TEST,PL=ABC,L=LIST)
22.14.12          (T,PL=ABC,L)
22.14.12               *
22.14.13 * POINTS TO FTN CONTROL CARD ERROR
```

The job proceeds with the option already processed or terminates and branches to an EXIT(S) card, depending upon the installation option.

I SOURCE INPUT PARAMETER

(Default I=INPUT)

I=lfn	lfn is the name of the file containing the source input. If I=INPUT is specified, source input is on the file INPUT. If I only is specified, source input is on the file COMPILE. If source input is on a file other than INPUT, the I=lfn form must be used. Compilation stops when an end-of- record **(end of section)** or end-of-file **(end of partition)** is encountered.

B BINARY OBJECT FILE

(Default B=LGO)

B	Generated binary object code is output on file LGO.
B=lfn	Generated binary object code is output on file lfn.
B=0	No binary object file is produced.
G=lfn BG=lfn GB=lfn G	Binary object file is loaded and executed at end of compilation

L LIST CONTROL

(Default L=OUTPUT, R=1)

y=lfn

The list control options specify the type of listing of the source program,y, and the file name, lfn, on which list output is to be written. If no list control options are specified, a listing is produced of the source program with informative and fatal diagnostics. If no file name is specified, OUTPUT is assumed.

y is any combination of one to four list control options selected from the letters: L,O,R,X,N. The letters must not be separated by commas. X and N cannot be specified at the same time.

lfn is the file on which output is to be written.

L=lfn	Source program, diagnostics, and short reference map listed (same as omitting list control parameter).
L	L defaults to L=OUTPUT.
L=0 or LR=0	Fatal diagnostics and the statements which caused them are listed. All other output, including intermixed COMPASS, is suppressed.
L=0,R=1	Level 1 reference map, fatal diagnostics and the statements which caused them listed. All other output is suppressed.
L=0,R=2	Level 2 reference map, fatal diagnostics and the statements which caused them listed. All other output is suppressed.
L=0,R=3	Level 3 reference map, fatal diagnostics and the statements which caused them listed. All other output is suppressed.
O=lfn	Generated object code is listed. The O option must not be used if the E option is selected.
R=lfn	Symbolic reference map is listed. R is included in the list control options for compatibility reasons only. Refer to R option in this section, and section 14 for full description of reference map.
X=lfn	A warning diagnostic is given for any non-ANSI usage. For example, if this option is selected and a 7- character symbolic name is used, (legal in FORTRAN Extended, but not defined under ANSI) the following warning diagnostic is printed: `7 CHARACTER SYMBOLIC NAME IS NON-ANSI`
N=lfn	Listing of informative diagnostics is suppressed; only diagnostics fatal to execution are listed.

For example, LRON = lfn specifies all options except non-ANSI diagnostics are to be listed foı the file lfn, and LO selects source program and generated object code listing on OUTPUT.

E EDITING PARAMETER

(Default E=COMPS)

E or E=lfn

Compiler generated object code is output as COMPASS card images for the SCOPE maintenance program UPDATE (refer to SCOPE Reference Manual). If E is omitted, normal binary object file is produced. The O and C options must not be specified if the E option is selected.

The object code output file starts with the card image: *DECK,name and ends with the card image: *END (name is the name of a program unit).

The object code output file lfn or COMPS is rewound and ready as UPDATE input. No binary file is produced. COMPASS is not called automatically. When the COMPS file is assembled S=FTNMAC must be specified on the COMPASS control card.

T ERROR TRACEBACK

(Default T omitted)

This option is provided to assist in debugging programs.

T	Calls to library functions are made with call-by-name sequence (section 10, part 3). Maximum error checking takes place, and full error traceback occurs if an error is detected.
T omitted	The more efficient (but less informative) call-by- value linkages are generated. Minimum error checking takes place, and no traceback is provided if an error is detected. A significant saving in memory space and execution time is realized.

Selecting the D parameter or OPT=0 automatically selects T.

ROUNDED ARITHMETIC SWITCH

(Default: arithmetic not rounded)

ROUND=op	op is an arithmetic operator: + - * / Single precision (real and complex) floating point arithmetic operations are performed using the hardware rounding features (refer to 6000 series and 7600 Computer Systems Reference Manuals). Any combination of the arithmetic operators can be specified. For example: ROUND= +-/

If this parameter is omitted (default condition), arithmetic computations are not rounded.

D DEBUGGING MODE PARAMETER

D or D=lfn

If the debug facility described in section 12 is used, D or D=lfn must be specified. This parameter automatically selects fast compilation (OPT=0) and full error traceback (T option). When the debug parameter is selected, any optimization level other than OPT=0 is ignored. A minimum field length of 61000 should be specified on the SCOPE control card if this option is selected.

lfn is the name of the file on which the user debug deck resides (figure 13-4, section 13). The default option for D=lfn is D=INPUT.

FTN(D) is equivalent to FTN(D=INPUT,OPT=0,T)

A EXIT PARAMETER

A	Compilation terminates and branches to an EXIT(S) control card if fatal errors occur during compilation. If there is no EXIT(S) control card, the job terminates.

Note: The S, GT and SYSEDIT parameters are of interest primarily to system programmers.

S SYSTEM TEXT FILE

(Default S=SYSTEXT)

S=lfn	Source of systems text information for intermixed COMPASS assemblies is on file lfn. If the only GT parameter is GT=0, the overlay named SYSTEXT is loaded. If parameter is omitted, information is on SYSTEXT overlay.
S=0	When COMPASS is called to assemble any intermixed COMPASS programs, it will not read in a system text file.
S=ovlname	The system text overlay, ovlname, is loaded from the job's current library set.
S=libname/ ovlname	The system text overlay, ovlname, is loaded from the library, libname. Libname can be a user library file or a system library.

GT GET SYSTEM TEXT FILE

GT=lfn	Loads the first system text overlay, if any, in the sequential binary file, lfn.
GT=lfn/ ovlname	Searches the sequential binary file, lfn, for a system text overlay with name ovlname, and loads the first such overlay encountered.
GT=0 or omitted	No system text is loaded from a sequential binary file.

A maximum of seven system texts can be specified. (Any combination of the GT, S and C parameters must not specify more than seven system texts.)

SYSEDIT SYSTEM EDITING

(Default SYSEDIT not selected)

This option is used mainly for system resident programs.

SYSEDIT	All input/output references are accomplished indirectly through a table search at object time. File names are not entry points in main program, and subprograms do not produce external references to the file name.

V SMALL BUFFERS OPTION

V	When this option is selected, the compiler uses 513-word buffers for its intermediate files. Programs with a large number of specifications are compiled with a smaller field length under this option. Since less space is used in the buffers, compile time may increase. If V is specified on a 7600 control card, it will be ignored.

C COMPASS ASSEMBLY

C	The COMPASS assembler is used to assemble the code generated by FTN. If C is omitted, the FTN assembler is used; it is two to three times faster than the COMPASS assembler. When the C parameter is specified, FTNMAC is supplied as additional text for the COMPASS assembly. Therefore, if the C option is selected, the maximum number of system texts which can be specified with the GT and S parameters is six.

R SYMBOLIC REFERENCE MAP (Default R=1)

R=n	Selects the kind of reference map required (section 14).
R=0	No map
R=1	Short map (symbols, addresses, properties)
R=2	Long map (symbols, addresses, properties, references by line number and a DO-loop map)
R=3	Long map with printout of common block members and equivalence groups

PL PRINT LIMIT (Default n=5000)

PL=n	n is the maximum number of records produced by the user program at execution time which can be written on the OUTPUT file. n does not include the number of records in the source program listing, and compilation and execution time listings; n ≤ 999 999 999
PL=nB	An octal number must be suffixed with a B; n ≤ 777 777 777B

Q PROGRAM VERIFICATION

Compiler performs full syntactic and semantic scan of the program and prints all diagnostics, but no object code is produced. A complete reference map is produced (with the exception of code addresses). This mode is substantially faster than a normal compilation; but it should not be selected if the program is to be executed. If Q is omitted, normal compilation takes place.

Z ZERO PARAMETER

Z	When Z is specified. all subroutine calls with no parameters are forced to pass a parameter list consisting of a zero word. This feature is useful only to COMPASS subroutines expecting a variable number of parameters (0 to 63). For example, CALL DUMP dumps storage on the OUTPUT file and terminates program execution. If no parameters are specified and Z is selected, a zero word parameter is passed. Z should not be specified unless necessary, as programs execute more efficiently if Z is omitted.

LCM LARGE CORE MEMORY ACCESS (Default LCM=D)

LCM=D	Selects 17-bit address mode for level 2 data. This method is the most efficient for generating code for data assigned to level 2. User LCM field length must not exceed 131,071 words.
LCM=I	Selects 21-bit address mode for level 2 data. This mode depends heavily upon indirect addressing. LCM=I must be specified if the user LCM field length exceeds 131,071 words. In neither case can a single common block be greater than 131,071 decimal words.

OPT OPTIMIZATION PARAMETER

OPT=m

m=0	Fast compilation (automatically selects T option)
m=1	Standard compilation and execution
m=2	Fast execution

The level of optimization performed by the compiler is determined by the value of m.

OPT=0	Compilation speed increases at expense of execution speed. (Selecting the D parameter automatically selects OPT=0.)
OPT=1	Normal compilation takes place.
OPT=2	Execution speed increases for certain loops. Two types of optimization are performed: Calculations which do not vary are removed from loops. Variables and constants from the body of a loop are assigned to registers.

The degree of optimization of DO and IF loops varies according to the following constraints:

> It must be the innermost loop (contain no loops).
>
> It 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 BUFFER IN/BUFFER OUT or ENCODE/DECODE statements. If input/output or any external calls occur, only calculations which do not vary are removed.
>
> Control must flow to the statement following the end of the IF loop when it completes.
>
> Entry into the IF loop must be through the sequence of statements preceding the start of the loop.

INVARIANT COMPUTATIONS

In many instances, a programmer codes calculations which do not change on successive iterations within a loop. When these computations are moved 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. However, if OPT=2 is specified the more efficient loop structure is produced by the compiler. A message is printed:

```
n WORDS OF INVARIANT RLIST REMOVED FROM
  THE LOOP STARTING AT LINE x
```

RLIST is the intermediate language of the compiler. The source language is translated first into RLIST, then into COMPASS. Optimization takes place during the RLIST phase, and it is at this point that invariant code is removed. The message notifies the programmer that his loop has been modified, and 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 produces 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 is removed from the loop regardless of the external call. In general, this process will occur unless R is a parameter to the external routine, or R is in common. When a variable is a member of an equivalence group, its use is not recognized as invariant if another member of the group is referenced inside the loop by non-standard subscripts. For standard subscripts, optimization will occur, although the assumption is made that all subscripting is within the bounds of array specifications. A standard subscript is one of the following forms; c and k are integer constants and v is an integer variable.

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

Subscript expressions which do not conform to the above are non-standard subscripts.

REGISTER ASSIGNMENT

For many loops, it is possible to keep commonly used variables and constants in the machine registers. Eliminating loads and stores from the body of the loop has two advantages:

The reduced number of loads and stores increases execution speed.

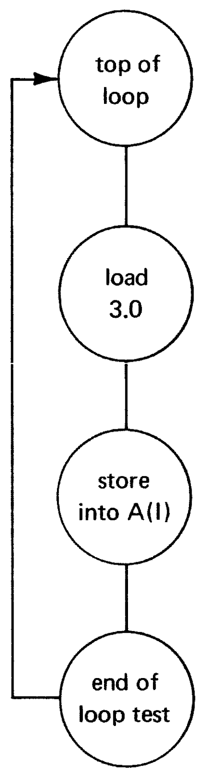
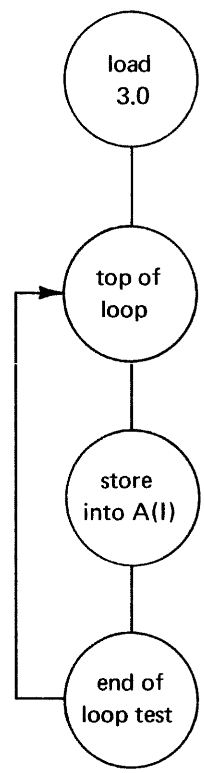
The loop is shortened and may fit in the instruction stack. A loop that fits in the instruction stack usually runs two to three times as fast as a comparable loop which does not fit in the stack.

Presently up to four X registers may be assigned over a loop. The 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:

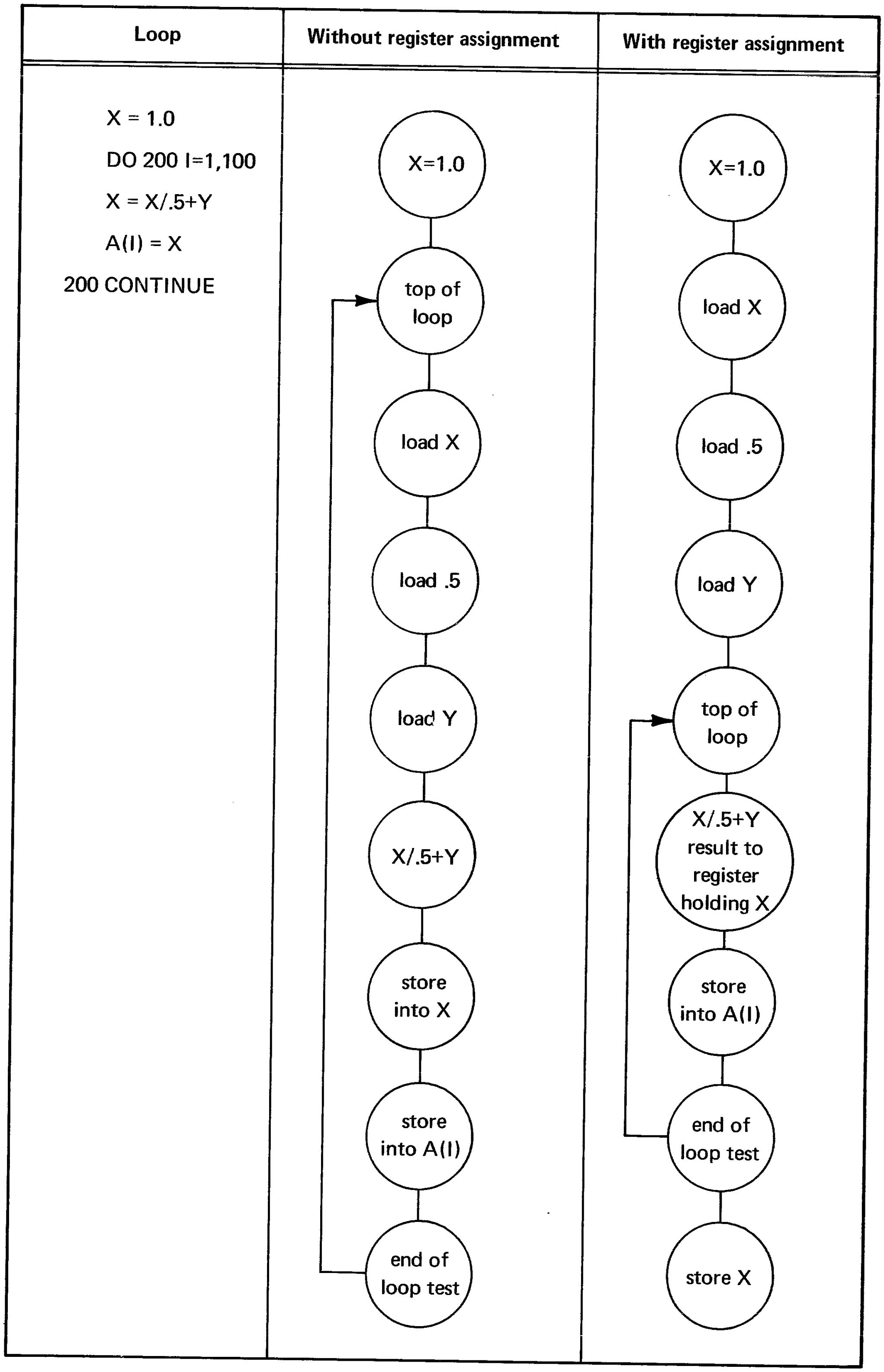
```
n REGISTERS ASSIGNED OVER THE LOOP BEGINNING AT LINE x
```

Register assignment will not be performed for loops containing external references.

Example:

Loop	Without register assignment	With register assignment
DO 100 I=1,2000 A(I) = 3.0 100 CONTINUE		

Example:



Loop
Without register assignment
With register assignment
X = 1.0
DO 200 I=1,100
X = X/.5+Y
A(I) = X
200 CONTINUE
X=1.0
top of loop
load X
load .5
load Y
X/.5+Y
store into X
store into A(I)
end of loop test
X=1.0
load X
load .5
load Y
top of loop
X/.5+Y result to register holding X
store into A(I)
end of loop test
store X

Example:

```
FTN (A,LRN,G,S=0)
```

Selects the following options:

A	Branch to EXIT(S) card if compilation errors occur
LRN	Source program, fatal diagnostics, and reference map are listed.
G	Generated binary object file is executed at end of successful compilation.
S=0	When COMPASS is called to assemble an intermixed COMPASS subprogram, it will not read in a systems text file.

Example:

```
FTN(G,T)
```

Source program on INPUT file, object code on LGO, normal listing on OUTPUT file, maximum error checking, no debug package, standard compile mode, and unrounded arithmetic. Program is executed if no fatal errors occur.

`FTN.` is equivalent to `FTN(I=INPUT,L=OUTPUT,B=LGO,S=SYSTEXT,R=1,OPT=1)`

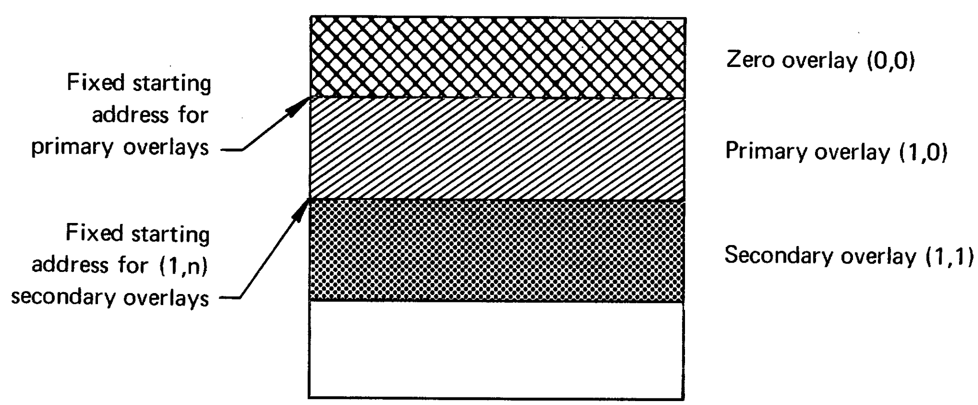
To reduce the amount of storage required, and to make more efficient use of his field length; a user can divide his program into overlays. Prior to execution, the sections of an overlay program are linked by the loader and placed on a mass storage device or tape file in their absolute form; no time is required for linking at execution time.

OVERLAYS

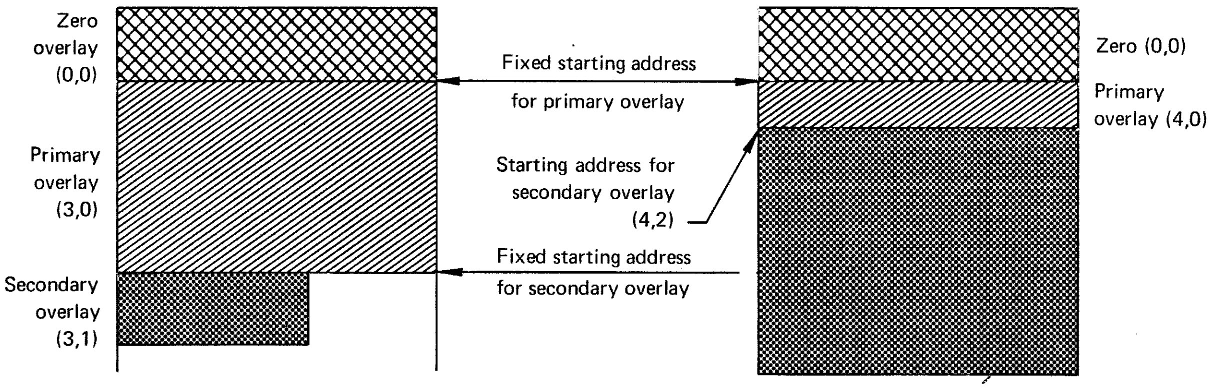
An overlay is a portion of a program written on a file in absolute form and loaded at execution time without relocation. As a result, the size of the resident loader for overlays can be reduced substantially. Overlays can be used when the organization of core can be defined prior to execution.

When each overlay is generated, the loading operation is completed by loading library and user subprograms and linking them together. The resultant overlay is in fixed format, in that internal references are fixed in their relationship to one another. The entire overlay has a fixed origin address within the field length and, therefore, is not relocatable. The overlay loader simply reads the required overlay from the overlay file and loads it starting at its pre-established origin in the user's field length.

Overlays are loaded into memory at three levels: zero, primary, and secondary.



The zero or main overlay is loaded first and remains in core at all times. A primary overlay may be loaded immediately following the zero overlay, and a secondary overlay immediately following the primary overlay. Overlays may be replaced by other overlays. For example, if a different secondary overlay is required, the overlay loader simply reads it from the overlay file and places it in memory at the same starting address as the previously loaded overlay.



When a primary overlay is loaded, the previously loaded primary overlay and any of its associated secondary overlays are destroyed. Loading a secondary overlay destroys a previously loaded secondary overlay. Loading any primary overlay destroys any other primary overlay. For this reason, no primary overlay may load other primary overlays.

Overlays are identified by a pair of integers:

zero or main overlay (0,0)

primary overlay (n,0)

secondary overlay (n,k)

n and k are positive integers in the range 0-77 octal. For any given program execution, all overlay identifiers must be unique.

For example, (1,0) (2,0) (3,0) (4,0) are primary overlays. (3,1) (3,2) (3,5) (3,7) are secondary overlays associated with primary overlay (3,0). Secondary overlays are denoted by the primary number and a non-zero secondary number. For example, (1,3) denotes that secondary overlay number 3 is related to primary overlay (1,0). (2,5) denotes secondary overlay 5 is related to primary overlay (2,0).

A secondary overlay can be called into core by its primary overlay or by the main overlay. Thus overlay (0,0) and overlay (1,0) may call (1,2); but overlay (2,0) may not call (1,2).

Overlay numbers (0,n) are not valid. For example, (0,3) is an illegal overlay number.

Execution is faster if the more commonly used subprograms are placed in the zero overlay, which remains in central memory/small core at all times, and the less commonly used subprograms are placed in primary and secondary overlays which are called into memory as required.

OVERLAY LINKAGES

The loader generates overlays and places them on a mass storage device or tape file in their absolute form. Linkage within an overlay is established during generation. The FORTRAN CALL statement (section 7) in a secondary overlay may call a subprogram within itself, or in its associated primary overlay, or in the zero overlay. Similarly, CALL statements in a primary overlay may call only subprograms within itself or in the zero overlay. Subprograms in the zero overlay may call only subprograms within the zero overlay. In order to call a primary or secondary overlay from a zero overlay, a CALL OVERLAY statement must be used.

An overlay may consist of one or more FORTRAN or COMPASS programs. The first program in the overlay must be a FORTRAN main program (not a subprogram). The program name becomes the primary entry point for the overlay when the overlay is called.

Data is passed between overlays through labeled or blank common. Any element of a labeled or blank common block in the main overlay (0,0) may be referenced by any higher level overlay. Any labeled or blank common declared in a primary overlay may be referenced only by the primary overlay and its associated secondary overlays–not by the zero overlay. If blank common is used for communicating between overlays, the user must ensure that sufficient field length is reserved to accommodate the largest loaded overlay in addition to blank common. Data stored in blank common must be used by each level of the overlay in exactly the same format, since no linkage is provided between the different levels of overlay and blank common at execution or load time.

Blank common is located at the top (highest address) of the first overlay in which blank common is declared. For example, if blank common is declared in the (0,0) overlay, it is located at the top of the (0,0) overlay and is accessible to all higher level overlays. If blank common is declared in the (1,0) overlay, it is allocated at the top of the (1,0) overlay and is accessible only to the associated (1,k) overlays. Labeled common blocks are generated in the overlay in which they are first encountered; data may only be preset in labeled common blocks in this overlay.

LCM common blocks must be defined and preset in the main (0,0) overlay. The entire overlay structure can reference an LCM common block.

CREATING AN OVERLAY

An overlay is established by an OVERLAY directive preceding the main program card for that overlay. An overlay consists of all programs appearing between its OVERLAY directive and the next OVERLAY directive or an end-of-file (6/7/8/9) card. The directive must be punched starting in column 7 or later and must be contained wholly on one card.

```
       7
       OVERLAY (fname,i,j,Cn)
```

file name — File name on which the generated overlay is to be written. All overlays need not reside on the same file.

i — Primary number, octal.

j — Secondary number, octal. (i and j must be 0,0 for the first overlay card.)

Cn — Optional parameter consisting of the letter C and a 6-digit octal number, which indicates the overlay is to be loaded n words from the start of blank common. Blank common is loaded after the zero overlay. With this method, the programmer can change the size of blank common at execution time. Cn cannot be included on the (0,0) overlay control card. If this parameter is omitted, the overlay is loaded in the normal way.

The first overlay directive must have a file name, subsequent directives may omit it, indicating that the overlays are related and are to be written on the same file.

Example:

```
OVERLAY(FNAME,0,0)
PROGRAM CAT(INPUT,OUTPUT,TAPE5=INPUT)
.
.
.
OVERLAY(1,0)
PROGRAM A
.
.
.
OVERLAY(1,1)
PROGRAM B
.
.
.
OVERLAY(1,2)
PROGRAM C
.
.
.
OVERLAY(1,3)
PROGRAM D
.
.
.
```

All the above overlays are written on the file FNAME.

Each OVERLAY directive must be followed by a PROGRAM statement. The PROGRAM statement for the zero or main overlay (0,0) must specify all file names such as INPUT, OUTPUT, TAPE1, etc., required for all overlay levels. File names should not appear in PROGRAM statements for other than the (0,0) OVERLAY.

Loading overlays from a file requires an end-around search of the file for the specified overlay; this can be time consuming in large files. When speed is essential, each overlay should be written on a separate file, or it should be called in the same order in which it was generated.

The group of relocatable decks processed by the loader must be presented to the loader in the following order. The main overlay must be loaded first. Any primary group followed by its associated secondary group can follow, then any other primary group followed by its associated secondary group, and so forth.

CALLING AN OVERLAY

A SCOPE control card causes the main (0,0) overlay to be loaded. Primary and secondary overlays are called by the following statement:

```
      7
      CALL OVERLAY (fname,i,j,recall,k)
```

fname	fname is the variable name of the location containing the name of the file (H format left justified display code) which includes the overlay if the k parameter is zero or is not specified. If a non-zero k parameter is specified, fname is the variable name of the location containing the overlay to be loaded.
i	Primary number of the overlay
j	Secondary number of the overlay
recall	Recall parameter. If 6HRECALL is specified, the overlay is not reloaded if it is already in memory. If the overlay is already in memory and the recall parameter is not used, the overlay is actually reloaded, thus changing the value of variables in the overlay.
k	k can be either an L format Hollerith constant of 7 characters, or any non-zero value. If k is a 7L... Hollerith constant, the overlay is loaded from the library named 7L... If k is any other non-zero value, the overlay is loaded from the global library set (refer to the SCOPE Reference Manual).

For example, the following statement causes a primary overlay to be loaded from the file named A:

```
CALL OVERLAY(1HA,1,0)
```

The following statement which specifies the k parameter as a non-zero value causes a main overlay, with the name BJR, to be loaded from the global library set.

```
CALL OVERLAY(3HBJR,0,0,0,1)
```

Numbers in the OVERLAY card are octal, thus to call OVERLAY (SAM,1,11) the statement CALL OVERLAY (3HSAM,1,9,0) or CALL OVERLAY (3HSAM,1,11B,0) must be used.

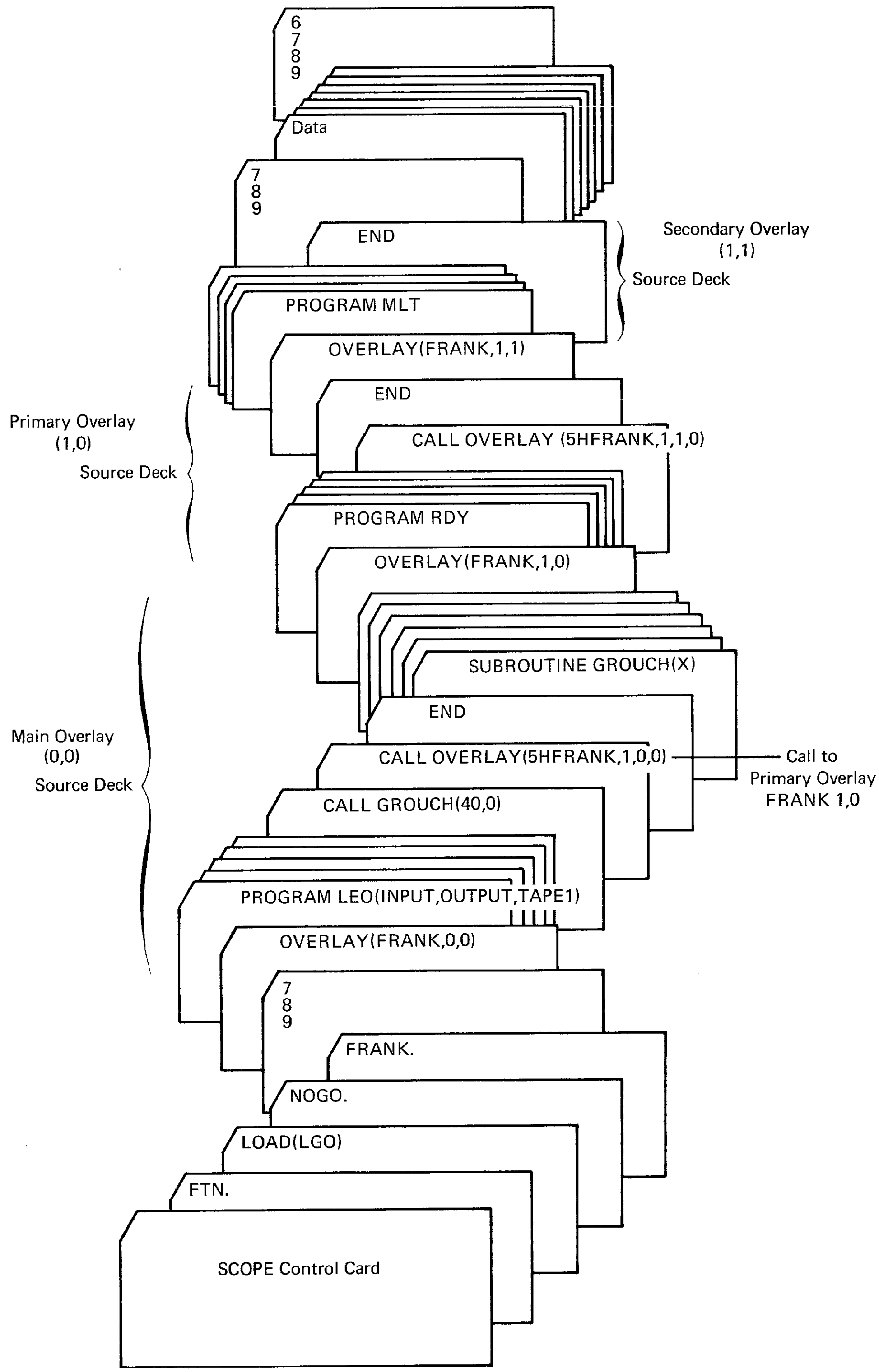
The three parameters, fname, i, and j must be specified; if any is omitted, a MODE error could result at execution time.

When an END statement is encountered in the main program of an overlay, control returns to the statement following the CALL OVERLAY statement.

Example:

```
OVERLAY(XFILE,0,0)
PROGRAM ONE(INPUT,OUTPUT,PUNCH)
.
.
.
CALL OVERLAY(5HXFILE,1,0,0)
.
.
.
STOP
END
OVERLAY(XFILE,1,0)
PROGRAM ONE ZERO
CALL OVERLAY(5HXFILE,1,1,0)
.
.
.
RETURN
END
OVERLAY(XFILE,1,1)
PROGRAM ONE ONE
.
.
.
RETURN
END
```

Example:



Preparation of Overlay 0,0; 1,0; and 1,1

The above example illustrates the preparation of zero primary and secondary overlays. The zero overlay, FRANK.0.0, consists of a main program LEO and a subroutine GROUCH. The primary overlay FRANK 1.0 consists of a main program MLT and a data deck. All three overlays reside on the file FRANK.

The SCOPE control card LOAD(LGO) requests the loader to load the program from the file LGO. As the loader reads file LGO, it encounters the overlay directive OVERLAY (FRANK,0,0) which instructs it to create a main overlay from the program and write it on file FRANK. When the absolute form of all the overlays has been generated, execution begins when the SCOPE control card FRANK. is encountered. FRANK. causes the main overlay to be loaded from file FRANK and executed.

During execution of the main overlay, the CALL OVERLAY (5HFRANK,1,0,0) statement is encountered and the primary overlay 1,0 is loaded into central memory. The CALL OVERLAY (5HFRANK,1,1) statement in the primary overlay causes the secondary overlay to be loaded into memory.

The primary and secondary overlays can reside on files other than FRANK. For example, the primary overlay could be on file JIM and the secondary overlay on file JOHN.

```
FTN.
LGO.
FRANK.
7/8/9
OVERLAY (FRANK,0,0)
PROGRAM LEO (INPUT,OUTPUT,TAPE1)
.
.
.
CALL OVERLAY (3HJIM,1,0,0)
.
.
.
OVERLAY (JIM,1,0)
PROGRAM RDY
.
.
.
CALL OVERLAY (4HJOHN,1,1,0)
END
OVERLAY (JOHN,1,1)
PROGRAM MLT
.
.
.
END
```

Example:

The following program, which contains several subroutines and functions, is to be used repeatedly. The entire program can be generated, therefore, as a main overlay and placed on the file in its absolute form. The SCOPE control card CATALOG creates a permanent file OVRLY where the absolute form of the program will be kept. When the program is required again the SCOPE permanent file OVRLY is called by an ATTACH control card.

The first program must be a main program; in this case program A.

SCOPE Control Cards

```
FTN.
LOAD(LGO)
CATALOG (OVRLY,REPEAT,ID=IBB)
NOGO.
7/8/9
```

Main Overlay

```
OVERLAY (REPEAT,0,0)
PROGRAM A (INPUT,OUTPUT,TAPE1)
.
.
.
END
SUBROUTINE B
.
.
.
END
FUNCTION C
.
.
.
END
SUBROUTINE D
.
.
.
END
REAL FUNCTION E
.
.
.
END
7/8/9
  data
6/7/8/9
```

Main program A and the subroutines and functions B-E reside on the file REPEAT in absolute form. They can be called and executed without recompilation by the SCOPE control cards:

```
SCOPE job card
ATTACH (OVRLY,REPEAT,ID=IBB)
REPEAT.
6/7/8/9
```

The SCOPE Reference Manual gives full details of the control cards which appear in the above program.

DEBUGGING FACILITY

The debugging facility allows the programmer to debug programs within the context of the FORTRAN language. Using the statements described in this section, the programmer can check the following:

Array bounds

Assigned GO TO

Subroutine calls and returns

Function references and the values returned

Values stored into variables and arrays

Program flow

The debugging facility, together with the source cross reference map, is provided specifically to assist the programmer develop or convert programs.

The debugging mode is selected by specifying D or D=lfn on the FTN control card (section 11). This control card parameter automatically selects fast compilation (OPT=0) and full error traceback (T option). If any other optimization level is specified, it will be ignored. The following examples are equivalent:

```
FTN (D)
FTN (D=INPUT,OPT=0,T)
FTN  (D,OPT=2)        OPT=2 is ignored, OPT=0 and T are automatically selected.
```

Debug output is written on the file DEBUG. When the job terminates, SCOPE gives the DEBUG file a print disposition and it is printed separately from the output file. To obtain debugging information on the same file as the source program, or any other file, DEBUG must be equivalenced to that file in the PROGRAM statement.

Examples:

```
PROGRAM EX (INPUT,OUTPUT,DEBUG=OUTPUT)
```

Debug output is interspersed with program output on the file OUTPUT.

```
PROGRAM EX(INPUT,OUTPUT,TAPEX,DEBUG=TAPEX)
```

Debug output is written on the file TAPEX.

The following control card sequence causes the debug output to be printed on the output file at termination of the job. It will not be interspersed with program output.

```
FTN(D)
LGO.
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
EXIT(S)                 Abnormal termination
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
```

When the debug mode is selected, programs execute regardless of compilation errors. Execution will, however, terminate at that point in the program where a fatal error is detected, and the following message will be printed:

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

Partial execution is prohibited for only three classes of errors.

Errors in specification statements

Missing DO loop terminators

Missing FORMAT statement numbers

Partial execution of programs containing fatal errors allows the programmer to insert debugging statements in his program to assist him in locating fatal and non-fatal errors.

When a program is compiled in debug mode, at least 12000 (octal) words will be required beyond the minimum field length for normal compilation. To execute, at least 2500 (octal) words beyond the minimum would be required. The CPU time required for compilation will also be greater than for normal OPT=0 compilation.

If the D option is not specified on the FTN control card, all debugging statements are treated as comments. Therefore, it is not necessary to remove the debugging statements after the program is sufficiently debugged.

All debugging options are activated and deactivated at compile time only. This compile time processing is not to be confused with program flow at execution time.

```
      PROGRAM TEST (OUTPUT,DEBUG=OUTPUT)
      .
      .
      .
      GO TO 4
      .
      .
      .
C$   (DEBUGGING OPTION)
C$   (DEBUGGING OPTION)
      .
      .
      .
    4 CONTINUE
      .
      .
      .
      END
```

Even though a section of code may never be executed, the debugging options are processed at compile time and are effective for the remainder of the program. In the above example, the code between the GO TO statement and the CONTINUE statement may never be executed. However, debugging statements between these statements are processed at compile time and are effective for the remainder of the program, or until deactivated by a C$ OFF statement.

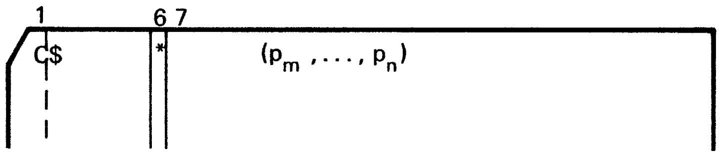
DEBUGGING STATEMENTS

```
 1           7
 C$          ds(p1,...,pn)
```

ds Type of option, beginning after column 6: DEBUG, AREA, ARRAYS, CALLS, FUNCS, GOTOS, NOGO, OFF, STORES, TRACE

p_i Argument list; details extent of the option, ds (not used with NOGO, GOTOS; required for AREA, STORES; optional for other options)

CONTINUATION CARD



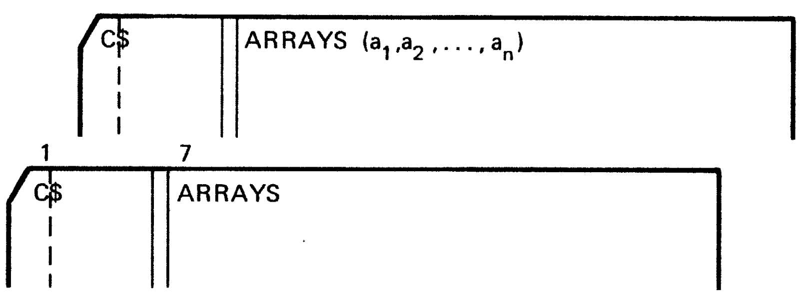
Debugging statements are written in columns 7-72, as in a normal FORTRAN statement, but columns 1 and 2 of each statement must contain the characters C$. Any character, other than a blank or zero, in column 6 denotes a continuation line. Columns 3, 4, and 5 of any debugging statement must be blank. The restriction on the number of continuation lines is the same as for FORTRAN continuation lines.

Comment cards may be interspersed with debugging statements. The statement separator ($) cannot be used with debugging statements. When the debug mode is not selected, all debugging statements are treated as comments.

Example:

```
C$    ARRAYS (A, BNUMB,Z10, C, DLIST, MATRIX,
C$   *NSUM, GTEXT,
C$   *TOTAL)
```

ARRAYS STATEMENT



$a_1,...,a_n$ array names

The ARRAYS statement initiates subscript checking on specified arrays. If no argument list specified, all arrays in the program unit are checked. Each time a specified or implied element of an array is referenced, the calculated subscript is checked against the dimensioned bounds. The address is calculated according to the method described in figure 2-1, section 2. Subscripts are not checked individually. If the address is found to be greater than the storage allocated for the array or less than one, a diagnostic is issued. The reference is then allowed to continue. Bounds checking is not performed for array references in input/ output statements, or in ENCODE/DECODE statements.

```
      PROGRAM ARRAYS (OUTPUT,DEBUG=OUTPUT)
      INTEGER  A(2), B(4), C(6), D(2,3,4)
      PRINT 1
    1 FORMAT(*0     ARRAYS  EXAMPLE*///)
*
*     TURN ON ARRAYS FOR ARRAYS  A  AND  D
*
C$    ARRAYS (A, D)
*
*     A(3) IS OUT OF BOUNDS  AND  ARRAYS IS ON FOR  A, SO A DIAGNOSTIC
*         IS PRINTED.
*
      A(3) = 1
*
*     B(5) IS OUT OF BOUNDS  BUT  ARRAYS IS NOT ON FOR  B, SO NO
*         DIAGNOSTIC IS PRINTED.
*
      B(5) = 1
*
      C(2) = A(A(3))
*
*     EVEN THOUGH A(3) WAS OUT OF BOUNDS, THE ASSIGNMENT TOOK PLACE.
*         A(A(3)) IS EQUIVALENT TO  A(1). THIS SUBSCRIPT IS IN BOUNDS,
*         HOWEVER THE REFERENCE TO A(3) WILL CAUSE A DIAGNOSTIC.
*
      D(-5,0,6) = 99
*
*     FOR THE ARRAY D(L,M,N) THE STORAGE ALLOCATED IS L * M * N.
*         THE ADDRESS OF THE ELEMENT D(I,J,K) IS COMPUTED AS FOLLOWS
*                  (I + L + * (J - 1 + M + (K - 1)))
*         FOR THE ELEMENT D(-5,0,6) THE SUBSCRIPT APPREARS TO
*         BE OUT OF BOUNDS BECAUSE THE INDIVIDUAL SUBSCRIPTS ARE OUT
*         OF BOUNDS.  HOWEVER, 23, THE COMPUTED ADDRESS, IS LESS THAN
*         24, THE STORAGE ALLOCATED, AND NO DIAGNOSED IS ISSUED.
*
*
*     TURN ON ARRAYS FOR ALL ARRAYS
*
C$    ARRAYS
*
*     WITH THIS FORM ALL ARRAY REFERENCES WILL BE CHECKED. THERE WILL
*         BE DIAGNOSTICS FOR B(5), C(-1), AND D(0,0,0).  BECAUSE A(2)
*         IS IN BOUNDS AND A(4) IS IN AN I/O STATEMENT, THERE WILL BE
*         NO DIAGNOSTICS FOR EITHER OF THESE REFERENCES.
*
      A(2) = 1
      B(5) = 2 + C(-1)
      D(0,0,0) = 1
      PRINT 2, A(4)
    2 FORMAT(1X, A10)
      END
```

ARRAYS EXAMPLE

```
/DEBUG/  ARRAYS  AT LINE   13- THE SUBSCRIPT VALUE OF      3 IN ARRAY A     EXCEEDS DIMENSIONED BOUND OF     2
/DEBUG/          AT LINE   20- THE SUBSCRIPT VALUE OF      3 IN ARRAY A     EXCEEDS DIMENSIONED BOUND OF     2
/DEBUG/          AT LINE   47- THE SUBSCRIPT VALUE OF      5 IN ARRAY B     EXCEEDS DIMENSIONED BOUND OF     4
/DEBUG/          AT LINE   47- THE SUBSCRIPT VALUE OF     -1 IN ARRAY C     EXCEEDS DIMENSIONED BOUND OF     6
/DEBUG/          AT LINE   48- THE SUBSCRIPT VALUE OF     -8 IN ARRAY D     EXCEEDS DIMENSIONED BOUND OF    24
```

CALLS STATEMENT

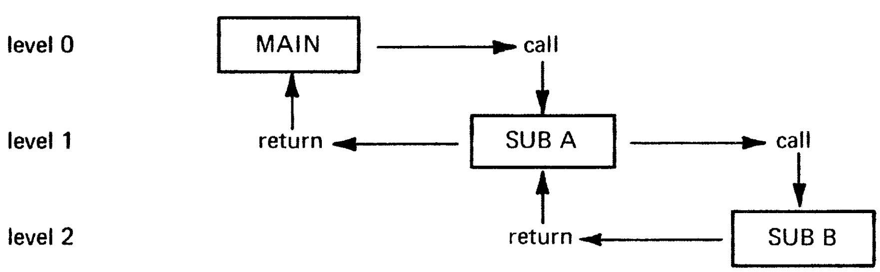
```
C$    CALLS (a1,...,an)
```

```
1     7
C$    CALLS
```

$a_1,....a_n$ subroutine names

The CALLS statement initiates tracing of calls to and returns from specified subroutines. If there is no argument list all subroutines will be traced. Non-standard returns, specified in a RETURNS list, are included. To trace alternate entry points to a subroutine, either the entry points must be explicitly named in the argument list, or the form with no argument list must be used (all external calls traced). The message printed contains the names of the calling and called routines, as well as the line and level number of the call and return.

A main program is at level zero; a subroutine or a function called by the main program is at level 1, another subprogram called by the subprogram at level 1, is at level 2, and so forth. Calls are shown in order of ascending level number, returns in order of descending level number.



For example, subroutine SUB A is called at level 1 and a return is made to level 0. SUB B is called at level 2 and a return is made to level 1.

Example:

```
      PROGRAM CALLS(OUTPUT,DEBUG=OUTPUT)
      PRINT 1
    1 FORMAT(*0      CALLS   TRACING*)
*
*     TURN ON CALLS FOR SUBROUTINES  CALLS1  AND  CALLS2
*
C$    CALLS(CALLS1, CALLS2)
      X = 1.
      CALL CALLS1 (X,Y), RETURNS (10)
   10 IF (X .EQ. 1.) CALL CALLS2
      CALL SUBNOT
      CALL CALLS1E (X,Y)
*
*     DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*         CALLS1 AND CALLS2.  SINCE THE CALLS ARE FROM THE MAIN PROGRAM,
*         THEY ARE AT LEVEL 0.  THE CALLS TO SUBNOT AND THE ALTERNATE
*         ENTRY POINT CALLS1E ARE NOT TRACED BECAUSE THEY DO NOT APPEAR
*         IN THE ARGUMENT LIST OF THE C$ CALLS STATEMENT.
*
*
*     TURN ON  CALLS FOR ALL SUBROUTINES
*
C$    CALLS
      CALL SUBNOT
      CALL CALLS2
      CALL CALLS1E (X,Y)
*     DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*         SUBNOT, CALLS2, AND CALLS1E, SINCE ALL CALLS ARE TO BE
*         TRACED.
      END


      SUBROUTINE CALLS1(X,Y), RETURNS(A)
      Y = -X
      IF (Y .NE. X) RETURN A
      RETURN
      ENTRY CALLS1E
      RETURN
      END


      SUBROUTINE CALLS2
      CALL CALLS1(X,Y), RETURNS(5)
    5 RETURN
      END


      SUBROUTINE SUBNOT
      X = -1.
      CALL CALLS1(X,Y), RETURNS(5)
    5 RETURN
      END
```

```
        CALLS   TRACING
  /DEBUG/  CALLS    AT LINE     9- ROUTINE CALLS1   CALLED   AT LEVEL     0
  /DEBUG/           AT LINE    10- ROUTINE CALLS1   RETURNS  TO LEVEL     0  AT STATEMENT 10
  /DEBUG/           AT LINE    10- ROUTINE CALLS2   CALLED   AT LEVEL     0
  /DEBUG/           AT LINE    11- ROUTINE CALLS2   RETURNS  TO LEVEL     0
  /DEBUG/           AT LINE    24- ROUTINE SUBNOT   CALLED   AT LEVEL     0
  /DEBUG/           AT LINE    25- ROUTINE SUBNOT   RETURNS  TO LEVEL     0
  /DEBUG/           AT LINE    25- ROUTINE CALLS2   CALLED   AT LEVEL     0
  /DEBUG/           AT LINE    26- ROUTINE CALLS2   RETURNS  TO LEVEL     0
  /DEBUG/           AT LINE    26- ROUTINE CALLS1E CALLED   AT LEVEL     0
  /DEBUG/           AT LINE    27- ROUTINE CALLS1E RETURNS  TO LEVEL     0
```

In this example, only calls from the main program are traced. To trace calls from subprograms, a C$ CALLS statement must appear in the subprograms.

FUNCS STATEMENT

```
C$    FUNCS (a1,...,an)
```

```
1     7
C$    FUNCS
```

If no function names (a_1,...,a_n) are listed, all external functions referenced in the program unit are traced. Alternate entry points must be named explicitly in the argument list, or implicitly in the C$ FUNCS statement with no paramenters.

Function tracing is similar to call tracing, but the value returned by the function is included in the debug message. Each time a specified external function is referenced, a message is printed which contains the routine name and line number containing the reference, function name and type, value returned and level number. The level concept is the same as for the CALLS statement.

Statement function references are not traced nor are function references in input/output statements.

Example:

The following program, VARDIM2, illustrates both the C$ FUNCS and C$ CALLS statements. All function references in the main program are traced because C$ FUNCS appears without an argument list; references to functions PVAL, AVG and MULT and the values returned to the main program (level 0) are traced. All subroutines in the main program are traced also because a C$ CALLS statement without an argument list appears.

Function references within the FUNCTION subprograms PVAL, AVG and MULT are traced since C$ FUNCS statements appear within these subprograms. If no C$ FUNCS statements appear in the subprograms, only main program function references will be traced.

```
      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
C     THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBPROGRAM CONCEPTS
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      PVALSF(X,Y) = PVAL(X,Y)
C$    CALLS
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
C
C     ALL EXTERNAL CALLS ARE DIAGNOSED.
C
C$    FUNCS
      AA = PVALSF(12,AVG)
      AM = PVALSF(12,MULT)
C
C     PVALSF IS A STATEMENT FUNCTION, SO THE FUNCS STATEMENT DOES NOT
C         APPLY TO IT AND NO MESSAGE IS PRINTED.  HOWEVER, THE EXTERNAL
C         FUNCTION PVAL IS REFERENCED WITHIN THE CODE FOR PVALSF,
C         AND THOSE REFERENCES ARE DIAGNOSED.
C     MULT AND AVG ARE NAMES AS ARGUMENTS TO PVALSF, HOWEVER, THE
C         FUNCTIONS ARE NOT ACTUALLY REFERENCED AND MESSAGES ARE NOT
C         PRINTED.
C
      STOP
      END
```

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
1     A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO2I=1,M
2     A(I)=A(I)+V
      RETURN
      END
```

```
      SUBROUTINE IOTA (A,M)
C     IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C          THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
1     A(I)=I
      RETURN
      END
```

```
      FUNCTION PVAL(SIZE,WAY)
C  PVAL COMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALUE IS RETURNED
C     BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED.  SIZE IS AN INTEGER
C     VALUE PASSED ON TO THE FUNCTION.
      INTEGER SIZE
C$    FUNCS(ABS)
      PVAL=ABS(WAY(SIZE))
C
C     WAY DOES NOT APPEAR IN THE ARGUMENT LIST FOR THE FUNCS STATEMENT,
C         SO ONLY THE REFERENCE TO ABS IS DIAGNOSED.
C
      RETURN
      END
```

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
      COMMON A(100)
      AVG=0.
      DO1I=1,J
1     AVG=AVG+A(I)
C$    FUNCS
C
C     ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
C
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
      REAL FUNCTION MULT(J)
C   MULT COMPUTES A STRANGE AVERAGE.  IT MULTIPLIES THE FIRST AND 12TH
C     ELEMENTS OF COMMON AND SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C     BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
C
      COMMON ARRAY(12)
C$    FUNCS
C
C     ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
C
      MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
      RETURN
      E N D
```

```
/DEBUG/  VARDIM2 AT LINE    8- ROUTINE SET     CALLED  AT LEVEL   0
/DEBUG/          AT LINE    9- ROUTINE SET     RETURNS TO LEVEL   0
/DEBUG/          AT LINE    9- ROUTINE IOTA    CALLED  AT LEVEL   0
/DEBUG/          AT LINE   10- ROUTINE IOTA    RETURNS TO LEVEL   0
/DEBUG/          AT LINE   10- ROUTINE INC     CALLED  AT LEVEL   0
/DEBUG/          AT LINE   11- ROUTINE INC     RETURNS TO LEVEL   0
/DEBUG/          AT LINE   15- REAL    FUNCTION PVAL   CALLED AT LEVEL   0
/DEBUG/  AVG     AT LINE   11- REAL    FUNCTION FLOAT  CALLED AT LEVEL   2
/DEBUG/          AT LINE   11- REAL    FUNCTION FLOAT  RETURNS A VALUE OF   12.00000000    AT LEVEL  2
/DEBUG/  PVAL    AT LINE    7- REAL    FUNCTION ABS    CALLED AT LEVEL   1
/DEBUG/          AT LINE    7- REAL    FUNCTION ABS    RETURNS A VALUE OF   1.500000000    AT LEVEL  1
/DEBUG/  VARDIM2 AT LINE   15- REAL    FUNCTION PVAL   RETURNS A VALUE OF   1.500000000    AT LEVEL  0
/DEBUG/          AT LINE   16- REAL    FUNCTION PVAL   CALLED AT LEVEL   0
/DEBUG/  MULT    AT LINE   11- REAL    FUNCTION AVG    CALLED AT LEVEL   2
/DEBUG/  AVG     AT LINE   11- REAL    FUNCTION FLOAT  CALLED AT LEVEL   3
/DEBUG/          AT LINE   11- REAL    FUNCTION FLOAT  RETURNS A VALUE OF   6.000000000    AT LEVEL  3
/DEBUG/  MULT    AT LINE   11- REAL    FUNCTION AVG    RETURNS A VALUE OF  -1.500000000    AT LEVEL  2
/DEBUG/  PVAL    AT LINE    7- REAL    FUNCTION ABS    CALLED AT LEVEL   1
/DEBUG/          AT LINE    7- REAL    FUNCTION ABS    RETURNS A VALUE OF   26.50000000    AT LEVEL  1
/DEBUG/  VARDIM2 AT LINE   16- REAL    FUNCTION PVAL   RETURNS A VALUE OF   26.50000000    AT LEVEL  0
```

STORES STATEMENT

```
1          7
C$         STORES (c1,c2,...,cn)
```

An argument list must be specified for the STORES statement.

$(c_1,...,c_n)$ are variable names or expressions in the forms:

variable name

variable name .relational operator. constant

variable name .relational operator. variable name

variable name .checking operator.

Relational operators are .EQ., .NE., .GT., .GE., .LT., .LE.

Checking operators are .RANGE., .INDEF., .VALID.

Example:

```
C$     STORES(SUM,DGAMP,AX,NET.LT.4,ROWSUM.RANGE.)

C$     STORES(A1,AGAIN,I,A2.EQ.5,IAGAIN.LE.IVAR)

C$     STORES(C.EQ.(1.,1.),L.VALID.,D.NE.10.004)

C$     STORES(G.RANGE.,TR.EQ..FALSE.)
```

The STORES statement is used to record changes in value of specified variables or arrays. The STORES statement applies only to assignment statements. Values changed as a result of input/output, or use in DATA, ASSIGN, COMMON, or argument lists to subroutines and functions are not detected. The STORES statement does not apply to the index variable in a DO loop.

If the value of a variable in an EQUIVALENCE group is changed, the STORES statement will not detect changes to the value of other variables in the group.

VARIABLE NAMES

In the first form of the STORES statement, a message is printed each time the value of a variable or an array element changes. The variable and name of the array must appear as arguments in the C$ STORES statement.

Example:

```
                  PROGRAM STORES (INPUT,OUTPUT,DEBUG = OUTPUT)
                  LOGICAL L1,L2
            C$    STORES (NSUM,DGAMP,AX)
                  NSUM = 20
 5                DGAMP = .5
                  AX = 7.2 + DGAMP
                  L1 = .TRUE.
                  L2 = .FALSE.
                  PLANT = 2.5
10                A = 7.5
                  PRINT 3
                3 FORMAT (1H0)
                  STOP
                  END
```

Each time the value of the variables NSUM, DGAMP and AX changes, a message is printed. The values of PLANT, A, L1 and L2 are not printed, since they do not appear in the argument list.

```
/DEBUG/  STORES  AT LINE   4- THE NEW VALUE OF THE VARIABLE NSUM    IS            20
/DEBUG/          AT LINE   5- THE NEW VALUE OF THE VARIABLE DGAMP   IS  .5000000000
/DEBUG/          AT LINE   6- THE NEW VALUE OF THE VARIABLE AX      IS  7.700000000
```

Array element names should not be specified in the parameter list of a STORES statement; the array name must be used. If an array element name appears, an informative diagnostic is printed.

Example:

```
      PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
      REAL A(10), B(4,2)
C$    STORES (A,B)
      B(1,2) = 5.5
      B(4,2) = 0.
      DO 4 N = 1,3
    4 A(N) = N+1
      PRINT 5
    5 FORMAT (1H0)
      STOP
      END
```

```
/DEBUG/  STORAR  AT LINE   4- THE NEW VALUE OF THE VARIABLE B    IS  5.500000000
/DEBUG/          AT LINE   5- THE NEW VALUE OF THE VARIABLE B    IS 0.
/DEBUG/          AT LINE   7- THE NEW VALUE OF THE VARIABLE A    IS  2.000000000
/DEBUG/          AT LINE   7- THE NEW VALUE OF THE VARIABLE A    IS  3.000000000
/DEBUG/          AT LINE   7- THE NEW VALUE OF THE VARIABLE A    IS  4.000000000
```

The values stored into array elements B(1,2) and B(4,2) appear in the debug output under the array name B in both cases, and array elements A(1), A(2), and A(3) appear under the array name A.

RELATIONAL OPERATORS

In the second form of the CS STORES statement, a message is printed only when the stored value satisfies the relation specified in the argument list.

```
      PROGRAM ST3 (INPUT,OUTPUT,DEBUG=OUTPUT)
    5 FORMAT (1H0)
      PRINT 5
      M = 5
C$    STORES (I.EQ.3,N.LE.M,ANT)
      I = 3
      I = 4
      N = 4
      N = 6
      J = 10
      ANT = 77.0
      END
```

```
/DEBUG/  ST3     AT LINE    6- THE NEW VALUE OF THE VARIABLE I     IS           3
/DEBUG/          AT LINE    8- THE NEW VALUE OF THE VARIABLE N     IS           4
/DEBUG/          AT LINE   11- THE NEW VALUE OF THE VARIABLE ANT   IS 77.00000000
```

I appears in the debug output when it is equal to 3; N appears when it is less than or equal to M. Since no relational operator is specified with ANT, it is printed whenever the value changes.

CHECKING OPERATORS

In the third form of the STORES statement, a message is issued only when the stored value is out of range, indefinite, or invalid as specified by the checking operator.

RANGE	Out of range
INDEF	Indefinite
VALID	Out of range or indefinite

For example:

```
C$    STORES (ROWSUM .RANGE., COLSUM .VALID.)
```

Whenever the value to be stored into ROWSUM is out of range, a message is printed. Whenever the value to be stored into COLSUM is out of range or indefinite, a message is printed.

HOLLERITH DATA

Hollerith data stored in a variable of type integer is interpreted by the STORES statement as an integer number. Hollerith data stored in a variable of type real or double precision is interpreted as a real or double precision number.

In the following example, the three integer variables IHOLL, IRIGHT and ILEFT contain the characters PA in display code (20 and 01).

```
IHOLL     20015555555555555555

          P A  blank fill

IRIGHT    00000000000000002001

          zero fill       P A

ILEFT     20010000000000000000

          P A zero fill
```

The Hollerith characters PA are interpreted as integer numbers. Since the values stored in IHOLL and ILEFT are greater than 2**48-1, an X is printed (section 9, Iw Output). The variable IRIGHT contains the value 2001 (octal) which is printed out by the STORES option in decimal as 1025.

The variable IHOLL is interpreted as a floating point number and its decimal value is printed.

Example:

```
          PROGRAM DEHOL (INPUT,OUTPUT,DEBUG=OUTPUT)
C$        DEBUG
C$        STORES(IHOL,IRIGHT,ILEFT,HOLL)
          IHOL=2HPA
          IRIGHT=2RPA
          ILEFT=2LPA
          HOLL=2HPA
          PRINT 1
        1 FORMAT (1H0)
          STOP
          END
```

```
/DEBUG/  DEHOL   AT LINE    6- THE NEW VALUE OF THE VARIABLE IHOL     IS              X
/DEBUG/          AT LINE    7- THE NEW VALUE OF THE VARIABLE IRIGHT   IS           1025
/DEBUG/          AT LINE    8- THE NEW VALUE OF THE VARIABLE ILEFT    IS              X
/DEBUG/          AT LINE    9- THE NEW VALUE OF THE VARIABLE HOLL     IS  .4021071096E+15
```

GOTOS STATEMENT

```
1     7
C$    GOTOS
```

No argument list must be specified with the C$ GOTOS statement. The GOTOS statement initiates checking of all assigned GO TO statements to ensure that the statement label assigned to the integer variable is in the GO TO statement list. If no match is found, a message is printed and transfer of control continues.

```
          PROGRAM GO TOS (OUTPUT,DEBUG=OUTPUT)
          INTEGER A
C$        GOTOS
*            (GOTOS NEVER USES AN ARGUMENT LIST)
*
          ASSIGN 1 TO A
          GO TO A (1, 2, 3)
*
*         IN THIS CASE NO MESSAGE IS PRINTED SINCE THE LABEL ASSIGNED TO
*            A IS IN THE GOTO LIST.
*
        4 PRINT 10
       10 FORMAT(*   --CONTROL TRANSFERED TO STATEMENT LABEL 4--*)
          STOP
        1 ASSIGN 4 TO A
          GO TO A (1, 2, 3)
*
*         IN THIS CASE A MESSAGE IS PRINTED SINCE THE LABEL 4 IS NOT IN
*            THE GOTO LIST.  CONTROL THEN TRANSFERS TO LABEL 4.
*
        2 CONTINUE
        3 CONTINUE
          END
```

```
/DEBUG/  GOTOS   AT LINE   16- ASSIGNED GOTO INDEX CONTAINS THE ADDRESS 002151. NO MATCH FOUND IN STATEMENT LABEL ADDRESS LIST
  --CONTROL TRANSFERED TO STATEMENT LABEL 4--
```

TRACE STATEMENT

```
C$    TRACE (lv)
```

```
1     7
C$    TRACE
```

lv is a level number 0-49. If lv = 0, tracing occurs only outside DO loops. If lv = n, tracing occurs up to and including level n in a DO nest. If no level is specified, tracing occurs only outside DO loops.

The C$ TRACE statement traces the following transfers of control within a program unit:

GO TO

Computed GO TO

Assigned GO TO

Arithmetic IF

True side of logical IF

Transfers resulting from a return specified in a RETURNS list are not traced. (These can be checked by the C$ CALLS statement.)

If an out-of-bound computed GO TO is executed, the value of the incorrect index is printed before the job is terminated.

Messages are printed each time control transfers during execution. The message contains the routine name, the line where the transfer took place, and the number of the line to which the transfer was made, as well as the statement number of this line, if present.

A message is printed each time control transfers at a level less than or equal to the one specified by lv. For example, if a statement C$ TRACE(2) appears before a sequence of DO loops nested four deep, tracing takes place in the two outermost loops only.

TRACE messages are produced at execution time, but TRACE levels are assigned at compile time; therefore, the compile time environment determines the tracing status of any given statement. For example, a DO loop TRACE statement applies only to control transfers occurring between the DO statement and its terminal statement at compile time (physically between the two in the source listing).

Example:

```
                              PROGRAM P(OUTPUT,DEBUG=OUTPUT)
                              DATA J/0/
      level 0        C$       TRACE(1)
                              IF (J .EQ. 0)  GO TO 11
 5    level 1              11 DO 1 I1  =  1, 3
                              IF ( (J+1) .EQ. I1 )  GO TO 12
                           12 J = 1
        level 2               DO 2 I2 = 1, 5
                              J = J + I2
10                            GO TO 2
                            2 CONTINUE
                     C$       TRACE(3)
        level 2               DO 20 I2 = 1, 3
                              IF ( I2 .EQ. 3 )  GO TO 20
15                            J = 2
          level 3             DO 3 I3 = 1, 4
                              IF ( J .GT. I3 )  GO TO 31
                           31 DO 4 I4 = 1, 2
            level 4           GO TO 4
20                          4 CONTINUE
                            3 CONTINUE
                           20 CONTINUE
                              J = 0
                            1 CONTINUE
25                            END
```

```
/DEBUG/  P     AT LINE    4- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE    4- CONTROL WILL BE TRANSFERRED TO STATEMENT 11    AT LINE    5
/DEBUG/        AT LINE    6- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE    6- CONTROL WILL BE TRANSFERRED TO STATEMENT 12    AT LINE    7
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/        AT LINE   14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/        AT LINE   14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
```

In the first level 2 loop no debug messages are printed since the TRACE(1) statement is in effect. However, when the TRACE(3) statement becomes effective, flow is traced up to and including level 3. There are no messages for transfers within the level 4 loop. To trace only inner loops, for example levels 3 and 4 in the above example, a C$ TRACE(4) statement is placed immediately before the DO statement for the level 3 loop (line 16). A C$ OFF (TRACE) statement is placed after the terminal line for the level 3 loop, so that subsequent program flow in levels 0, 1, and 2 is not traced.

The level number applies to the entire program unit; it is not relative to the position of the C$ TRACE statement in the program. For example, to trace the level 4 DO loop in Program P

```
     C$   TRACE(4)
```

must be specified. Positioning the statement C$ TRACE(1) before statement 31 would not achieve the same result.

Care must be taken with the use of debugging statements within DO loops. Since nested loops are executed more frequently, the quantity of debug output may quickly multiply.

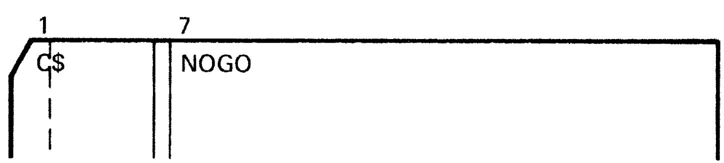
The C$ TRACE (lv) statement traces transfers of control within DO loops. However, transfers between the terminal statement and the DO statement are not traced.

Example:

```
          DO 100 I = 1,10
          .
          .
          .
      100 CONTINUE
```

Transfers from statement 100 to the DO statement are not traced.

NOGO STATEMENT



No argument list must be specified with this statement. The NOGO statement suppresses partial execution of a program containing compilation errors.

When the debug mode is specified and the NOGO statement is not present, programs execute regardless of compilation errors until a fatal error is encountered.

If a NOGO statement is present anywhere in the program, it applies to the entire program. It is therefore not affected by an OFF statement or by bounds in an AREA statement.

DEBUG 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. Interspersed debugging statements (figure 13-1) change the FORTRAN generated line numbers for a program.

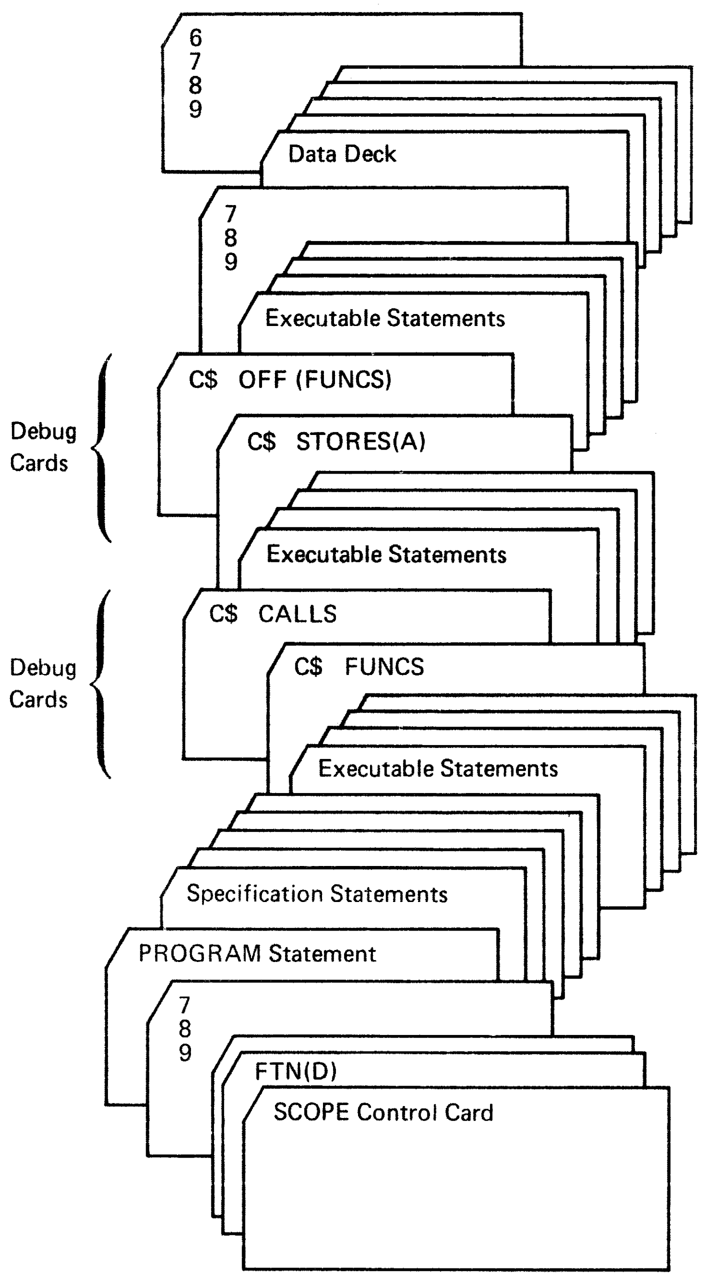
Debugging statements also may be grouped to form a debugging deck in one of the following ways:

As a deck placed immediately after the PROGRAM, SUBROUTINE or FUNCTION statement heading the routine to which the deck applies (internal debugging deck, figure 13-3). Any names specified in the DEBUG statement, other than the name of the enclosing routine, are ignored.

As a deck immediately preceding the first source deck in the job INPUT file (external debugging deck, figure 13-2).

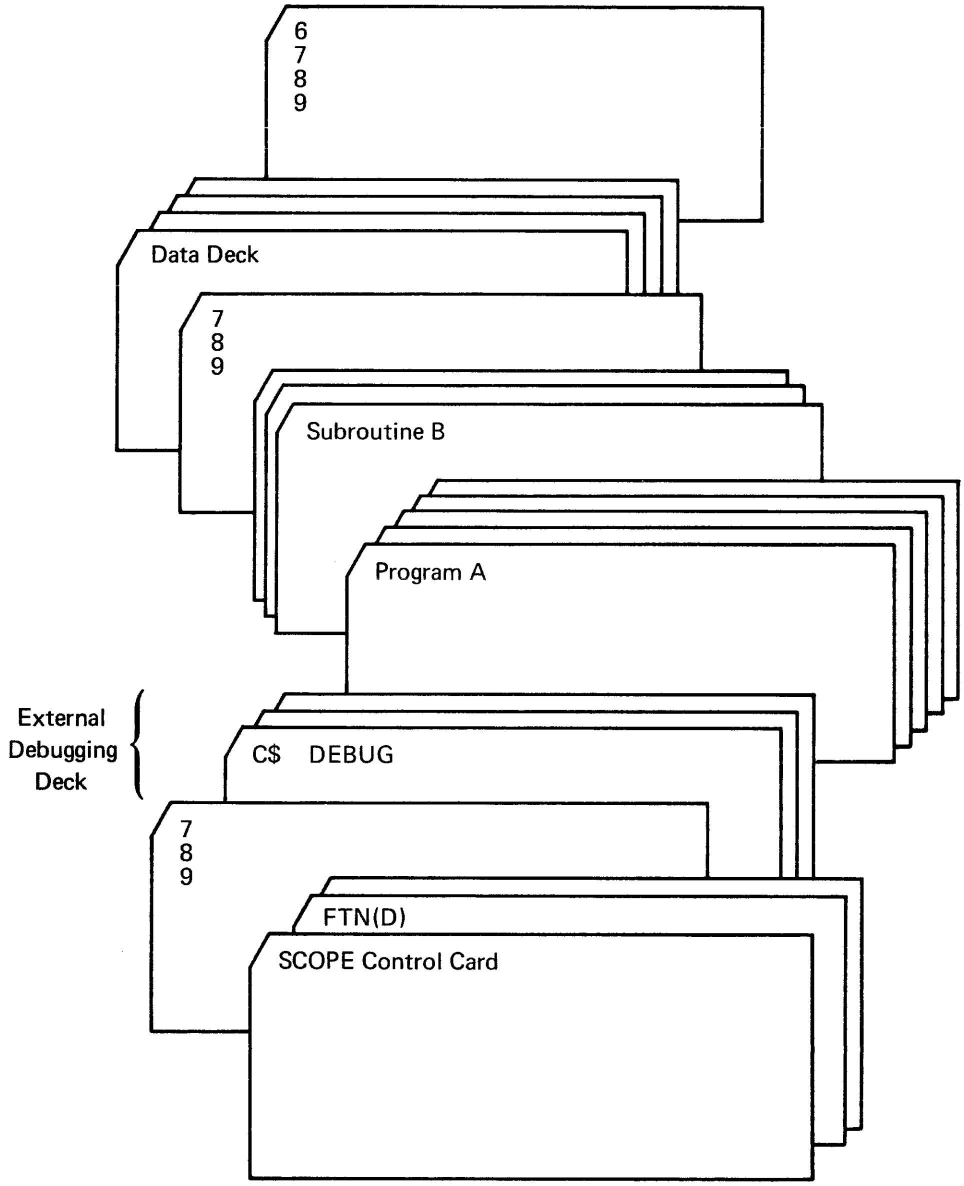
As one or more decks on the file specified by the D parameter on the FTN control card (external debugging deck, figure 13-4). When no name is specified by the D parameter, the INPUT file is assumed.

All debugging decks must be headed by a C$ DEBUG card. In an internal debugging deck, the C$ DEBUG card is used without an argument list, since the deck can only apply to the routine in which it is inserted. In an external debugging deck, a C$ DEBUG may be used with or without an argument list. The statements in the external debugging deck apply to all program units in the compilation.



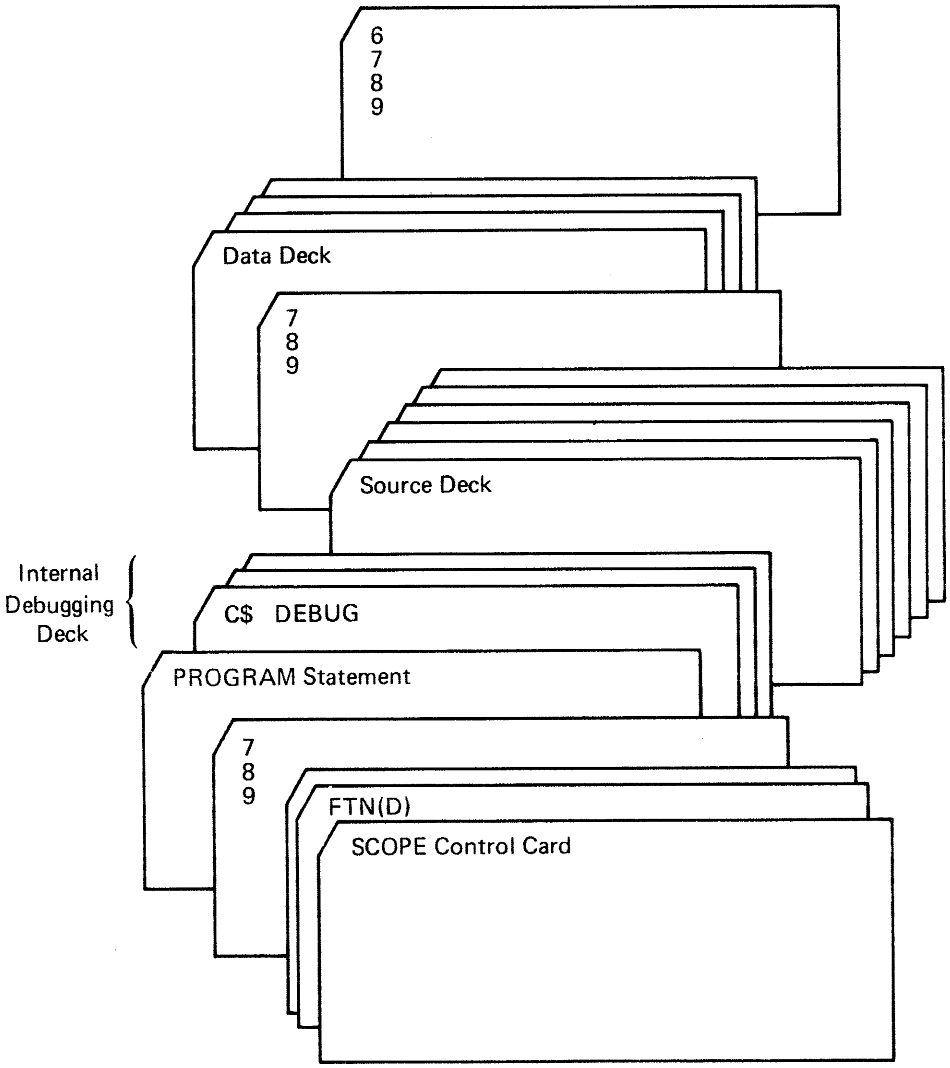
Debugging cards are interspersed; they are inserted at the point in the program where they will be activated.

Figure 13-1. Example of Interspersed Debugging Statements



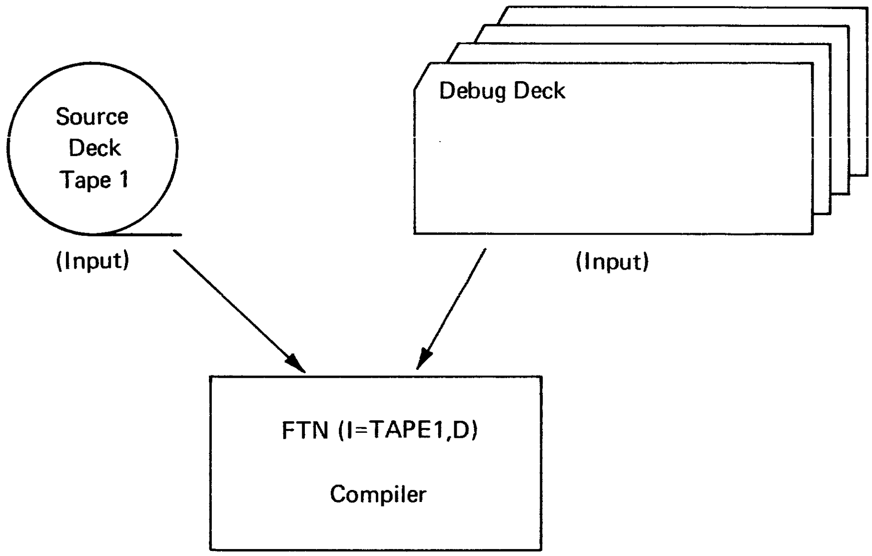
The external debugging deck is placed immediately in front of the first source line. All program units (here, Program A and Subroutine B) will be debugged (unless limiting bounds are specified in the deck). This positioning is particularly useful when a program is to be run for the first time, since it ensures that all program units will be debugged.

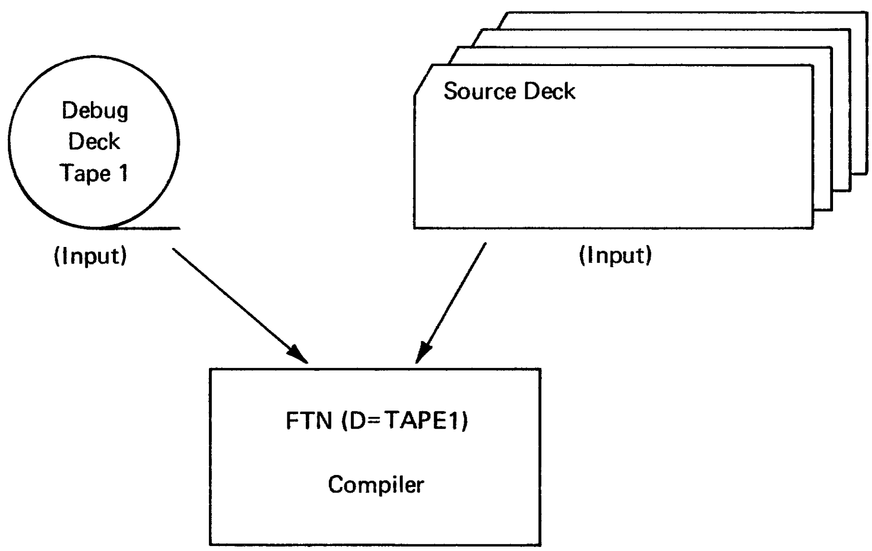
Figure 13-2. External Debugging Deck



When the debugging deck is placed immediately after the program name card and before any specification statements, all statements in the program unit will be debugged (unless limiting bounds are specified in the deck); no statements in other program units will be debugged. This positioning is best when the job is composed of several program units known to be free of bugs and one unit that is new or known to have bugs.

Figure 13-3. Internal Debugging Deck





The debugging deck is placed on a separate file (external debugging deck) named by the D parameter on the FTN control card and called in during compilation. All program units will be debugged (unless the program units to be debugged are specified in the deck). This positioning is useful when several jobs can be processed using the same debugging deck.

Figure 13-4. External Deck on Separate File

DEBUG STATEMENT

```
C$     DEBUG

1      7
C$     DEBUG (name1 ,... , namen)
```

name$_1$,...,name$_n$ routines to which the debugging deck applies

Internal and external debugging decks start with a DEBUG statement and end with the first card other than a debugging statement or comment. Interspersed debugging statements do not require a DEBUG statement.

In an internal debugging deck, the first form C$ DEBUG statement without an argument list is generally used, since the deck can apply only to the program unit in which it appears. If a name is specified it must be the name of the routine containing the debugging deck; if any other name is specified, an informative diagnostic is printed.

In an external debugging deck, if no names are specified, the deck applies to all routines compiled. Otherwise, it will apply to only those program units specified by name$_1$,...,name$_n$; if any other name is specified, an informative diagnostic is printed.

Example:

In the following program, a DEBUG statement is not required since the debugging statement, C$ STORES (A,B), is interspersed.

```
      PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
      REAL A(10), B(4,2)
C$    STORES (A,B)
      B(1,2) = 5.5
      B(4,2) = 0.
      DO 4 N = 1,3
    4 A(N) = N+1
      PRINT 5
    5 FORMAT (1H0)
      STOP
      END
```

However, if the C$ STORES statement follows the PROGRAM statement, this is an internal debugging deck, and a C$ DEBUG statement must appear.

```
      PROGRAM DEHOL (INPUT,OUTPUT,DEBUG=OUTPUT)
C$    DEBUG
C$    STORES(IHOL,IRIGHT,ILEFT,HOLL)

      IHOL=2HPA
      IRIGHT=2RPA
      ILEFT=2LPA
      HOLL=2HPA
      PRINT 1
    1 FORMAT (1H0)
      STOP
      END
```

There can be several DEBUG statements in an external deck, and a routine can be mentioned more than once.

```
C$    DEBUG
C$    STORES(I,J)
C$    DEBUG(MAIN,EXTRA,NAMES)
C$    ARRAYS(VECTAB,MLTAB)
C$    DEBUG(MAIN)
C$    TRACE
C$    CALLS(EXTRA,NAMES)
```

AREA STATEMENT

```
1     7
C$    AREA bounds1,...,bounds n

C$    AREA/name1/bounds1,...,bounds n,.../name n/bounds1,...,bounds n
```

C$ AREA($bounds_1$,...,$bounds_n$) is used in internal debugging decks only.

$name_1$,$name_2$,...,$name_n$ are the names of routines to which the following bounds apply.

bounds are line positions defining the area to be debugged.

bounds can be written in one of the following forms:

(n_1,n_2)	n_1	Initial line position
	n_2	Terminal line position
(n_3)	n_3	Single line position to be debugged
(n_1,*)	n_1	Initial line position
	*	Last line of program
(*,n_2)	*	First line of program
	n_2	Terminal line position
(*,*)	*	First line of program
	*	Last line of program

Line positions can be:

nnnnn	Statement label
Lnnnn	Source program line number as printed on the source listing by the FORTRAN Extended compiler (source listing line numbers change when debugging cards are interspersed in the program).
id.n	Alphanumeric UPDATE line identifier (refer to SCOPE Reference Manual); id must begin with an alphabetic character and contain no special characters.

A comma must be used to separate the line positons, and embedded blanks are not permitted. Any of the line position forms may be combined and bounds may overlap.

The AREA statement is used to specify an area to be debugged within a program unit. All debugging statements applicable to the program areas designated by the AREA statement must follow that statement. Each AREA statement cancels the preceding program AREA statement. An AREA statement (or contiguous set of AREA statements) specifies bounds for all debugging statements that occur between it and the next C$ DEBUG, AREA statement, or FORTRAN source statement.

AREA statements may appear only in an external or an internal debugging deck (figures 13-2, 13-3, and 13-4). If they are interspersed in a FORTRAN source deck, they will be ignored.

In an external debugging deck, the following form, with /name$_i$/ specified, must be used. It can be used with both forms of the DEBUG statement.

```
C$    DEBUG
1     7
C$    AREA/name1/bounds1,...,boundsn,.../namen/bounds1,...,boundsn
```

or

```
C$    DEBUG (name1,...,namen)
1     7
C$    AREA/name1/bounds1,...,boundsn,.../namen/bounds1,...,boundsn
```

If /name$_i$/ is omitted, or names in the /name$_i$/ list do not appear in (name$_1$,...,name$_n$) in the DEBUG statement, the AREA statement is ignored.

In an internal debugging deck, the following form is used, and the bounds apply to the program unit that contains the deck.

```
C$    DEBUG
1     7
C$    AREA bounds1,...,boundsn
```

Example:

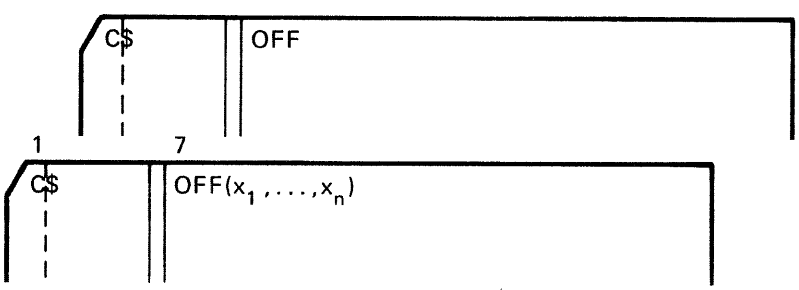
External deck

```
C$    DEBUG
C$    AREA/PROGA/(XNEW.10,XNEW.30),/SUB/*,L50)
C$    ARRAYS (TAB,TITLE,DAYS)
C$    AREA/SUB/(15,99)
C$    STORES (DAYS)
```

Internal deck

```
C$    DEBUG
C$    AREA (L10,*)
C$    FUNCS (ABS)
```

OFF STATEMENT



$x_1,...,x_n$ debug options

The OFF statement deactivates the options specified by x_i or all currently active options except NOGO, if no argument list exists. Only options activated by interspersed debugging statements are affected. Options activated in debug decks or by subsequent debugging statements are not affected.

The OFF statement is effective at compile time only. In a debugging deck, the OFF statement is ignored.

Example:

```
      PROGRAM OFF (OUTPUT,DEBUG=OUTPUT)
C$    DEBUG
C$    STORES(C)
      INTEGER A, B, C
C$    STORES(A, B)

      A = 1
      B = 2
      C = 3
*
*     MESSAGES WILL BE PRINTED FOR STORES INTO A, B, AND C.
*
C$    OFF
*
      A = 4
      B = 5
      C = 6
*     THE OFF STATEMENT WILL ONLY AFFECT THE INTERSPERSED DEBUGGING
*         STATEMENT, SO THERE WILL BE NO MESSAGES FOR STORES INTO
*         A OR B.  HOWEVER, C$  STORES(C) IN THE DEBUGGING DECK IS NOT
*         AFFECTED, AND A MESSAGE IS PRINTED FOR A STORE INTO C.
*
      END
```

```
/DEBUG/   OFF     AT LINE    7- THE NEW VALUE OF THE VARIABLE A       IS              1
/DEBUG/           AT LINE    8- THE NEW VALUE OF THE VARIABLE B       IS              2
/DEBUG/           AT LINE    9- THE NEW VALUE OF THE VARIABLE C       IS              3
/DEBUG/           AT LINE   17- THE NEW VALUE OF THE VARIABLE C       IS              6
```

PRINTING DEBUG OUTPUT

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

At object time, printing is performed by seven debug routines coded in FORTRAN. These routines are called by code generated at compile time when debugging is selected.

Routine	Function
BUGARR	Checks array subscripts
BUGCLL	Prints messages when subroutines are called and when return to calling program occurs
BUGFUN	Prints messages when functions are called and when return to calling program occurs
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

STRACE

Traceback information from a current subroutine level back to the main level is available through a call to STRACE. STRACE is an entry point in the object routine BUGCLL. A program need not specify the D option on the FTN card to use the STRACE feature.

STRACE output is written on the file DEBUG; to obtain traceback information interspersed with the source program DEBUG should be equivalenced to OUTPUT in the PROGRAM statement.

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
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
```

A main program is at level 0; a subroutine or function called by the main program is at level 1; another subprogram called by a subprogram is at level 2, etc. Calls are shown in order of ascending level number, returns in order of descending level number.

For additional information regarding the debugging facility, refer to the FORTRAN Extended Debug User's Guide.

SYMBOLIC REFERENCE MAP

The reference map, which appears on a separate page following the source listing of the program, is a debugging aid useful for detecting errors which do not show up as compilation errors, such as, names which have been keypunched incorrectly. Program flow can be traced and the structure of the program examined.

The reference map is a list of programmer created symbols in a program unit. Names of symbols generated by the compiler (such as library routines called for input/output) do not appear. Names are listed alphabetically within the following classes:

> Entry points, variables, file names, external references, inline functions, NAMELIST group names, statement and FORMAT statement numbers, DO loops, common blocks, equivalence classes.

The programmer may select from three types of reference map or suppress the map completely. The type of map produced is determined by the R option on the FTN control card.

R = 0	No map
R = 1	Short map (symbols, addresses, properties)
R = 2	Long map (symbols, addresses, properties, references by line number, and DO loop map)
R = 3	Long map and printout of common block members and equivalence classes

If no selection is made, the default option is R = 1. However, if the control card option L, which suppresses output, equals 0; no map is produced.

Field length should be increased by 1000 octal if the long map is specified; if it is too short, one of the following error messages is printed:

```
CANT SORT THE SYMBOL TABLE              INCREASE FL  BY nnnB

REFERENCES AFTER LINE nnn LOST          INCREASE FL  BY nnnB
```

Fatal errors in the source program cause certain parts of the map to be suppressed, incomplete, or inaccurate. The DO loop map is suppressed, and assigned addresses are different when fatal execution or compilation errors occur. References are not accumulated for statements containing syntax errors.

The number of references that can be accumulated and sorted for the reference map is eight times the number of symbols in the source program if the field length is 50000 octal. For example, in a source program containing 1000 (decimal) symbols, with a field length of 50000 octal, approximately 8000 (decimal) references can be accumulated.

Although formats for portions of the reference map differ, they all contain the following information:

The symbol as it appears in the FORTRAN program

Properties associated with the symbol

List of references to the symbol; line numbers in the list refer to the first line of the statement. Multiple references on a line are printed as n*k where n is the number of references and k is the line number.

CLASSES

ENTRY POINTS

Entry point symbols include subprogram names and names appearing in ENTRY statements; they are printed in the reference map under the headings: ENTRY POINTS, DEF LINE, REFERENCES.

ENTRY POINTS	Entry point name as it appears in the source program, and the program relative address.
DEF LINE	Line number of subprogram statement on which entry point is defined.
REFERENCES	Line number at which entry point is referenced (none for main program). RETURN statements constitute a reference to an entry point. References to a function value, in a function subprogram, appear in the variable map.

```
             SUBROUTINE IOTA (A,M)
      C      IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
      C          THE ARRAY A
             DIMENSION A(M)
             DO1I=1,M
           1 A(I)=I
             RETURN
             END
```

```
ENTRY POINTS      DEF LINE      REFERENCES
   2  IOTA            1             7
```

VARIABLES

Variable symbols include variables and arrays (including those in common), dummy arguments, RETURNS names, and for a function subprogram, the function name when it is used as a variable.

```
VARIABLES     SN  TYPE          RELOCATION
    0  A          RETURNS                          0  X          REAL                F.P.
    0  Y          REAL              F.P.
```

VARIABLES	Object program or common relative address; 0 for dummy arguments, and variable name as it appears in source program.
SN	Stray name flag. Variable names which appear only once in a subprogram are indicated by *. They are classified as stray, since they may be keypunch errors, misspellings, etc. A legal usage that would cause a name to be called stray is a DO loop in which the control variable is not referenced. (DO loops are mapped for R = 2 and R = 3 only.)
TYPE	Type of variable (logical, integer, real, double precision, complex). RETURNS is printed for RETURNS dummy arguments.
RELOCATION	Subdivided into properties, blockname, and references.
Properties	The keywords UNDEF, ARRAY, UNUSED may be printed in this column.
*UNDEF	Symbol has not been defined. Variables used before definition are not listed as undefined. For reference map purposes, a symbol is considered to be defined if any of the following conditions hold:

It appears in a COMMON or DATA statement.

It is a member of an equivalence group other than the base member (The base member of an equivalence group is the member with the smallest address. In an array X(10), X(1) has the smallest object program address, X(10) the largest.) An undefined non-base member is not detected by the compiler.

It is a simple variable or array element on the left-hand side of an assignment statement.

It appears in an ASSIGN statement.

It is the control variable in a DO loop.

It is a simple variable or array element which appears as an argument in a subroutine or function call.

It appears in an input/output list.

It is a dummy argument.

ARRAY	Symbol is an array name.
*UNUSED	Symbol is an unused dummy argument. If no further information appears on the line and it is not a RETURNS argument, it is a simple variable.

Blockname

blank	Symbol is not in common; address is relative to program unit.
F.P.	Dummy argument.
//	Symbol is in blank common; address is relative to blank common.
name	Name of labeled common block where symbol appears.

References

REFS	Number of times variable names appear in specification or assignment statements.

DEFINED	Number of times names are defined. Definitions are listed for names appearing in DATA statements, control variables of DO loops, names defined in an ASSIGN or assignment statement, and names defined by READ or ENCODE/DECODE statements. Dummy arguments are defined in the subprogram header line.
I/O REFS	Input/output references are collected for symbols used as variable file names in input/output statements.

References to the function name in a function subprogram are listed in the VARIABLE map rather than the EXTERNALS map.

References are collected after statement functions are expanded; they are not collected for the arguments before the 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.

Example:

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF CCMMON.
      COMMON A(100)
      AVG=0.
      DO 1 I = 1,J
   1  AVG=AVG+A(I)
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
         SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS
    2   AVG

VARIABLES      SN  TYPE            RELOCATION
    0   A           REAL        ARRAY    / /                 53   AVG          REAL
   54   I           INTEGER                                   0   J            INTEGER             F.P.

EXTERNALS          TYPE     ARGS
        FLOAT       REAL        1

STATEMENT LABELS
    0   1

COMMON BLOCKS    LENGTH
        / /          100

STATISTICS
  PROGRAM LENGTH      57B        47
    BLANK COMMON     144B       100
```

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 statement in a main program. They are printed in the reference map under the following headings: FILE, NAME, MODE.

FILE	Object program relative address of the file information table (FIT)
NAME	Name of file
MODE	Type of input/output operations performed. They may be formatted (FMT), unformatted (UNFMT), buffered (BUF), or MIXED (combination of FMT, UNFMT, BUF).

References are divided into categories:

READS	Input operations
WRITES	Output operations
MOTION	Positioning operations; rewind, backspace, and ENDFILE

```
PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)


FILE NAMES           MODE
     0  DEBUG                      0  OUTPUT                          0  TAPE6
```

EXTERNAL REFERENCES

External references include names of subroutines or functions external to a program unit. If the T or D option is specified on the FTN control card, intrinsic functions are compiled as external references. Library functions appear as external references when T or D is specified, as they are called by name. Names of system routines not explicitly called in the source program, such as those used for input/output and exponentiation, are suppressed.

External references are printed in the reference map under the following headings: EXTERNALS, TYPE, ARGS.

EXTERNALS	Symbol as it appears in source program.
TYPE	
blank	Subroutine
NO TYPE	Conversion follows the same rule as for octal or Hollerith data
other	Real, integer, double precision, complex, logical
ARGS	Number of arguments used to reference the external symbol, followed by:
blank	Programmer defined function or subroutine
F.P	Dummy argument
LIBRARY	Call by value library function

The line numbers on which symbol was referenced appear as the last item.

Example:

```
EXTERNALS           TYPE     ARGS
        AVG                   0                 INC              3
        IOTA                  2                 MULT             0
        PVAL       REAL       2                 SET              3
```

INLINE FUNCTIONS

Inline functions include names of intrinsic and statement functions appearing in the subprogram. They are printed under the following headings: INLINE FUNCTIONS, TYPE, ARGS, INTRIN, SF, LINE, REFERENCES.

```
INLINE FUNCTIONS    TYPE     ARGS
        PVALSF     REAL        2     SF
```

INLINE FUNCTIONS	Symbol name as it appears in the listing
TYPE	Arithmetic type, NO TYPE means no conversion is performed in mixed mode expressions
ARGS	Number of arguments with which the function is referenced
INTRIN	Intrinsic function
SF	Statement function
LINE	Blank for intrinsic functions; definition line for statement functions
REFERENCES	Lines on which function is referenced

NAMELIST GROUP NAMES

The NAMELIST group name, line on which it was defined, and all references to the name are included in this class. If NAMELIST names were not used, this portion of the reference map is suppressed.

Example:

NAMELISTS	DEF LINE	REFERENCES			
SAMPLE	2	3	6	10	11
TEST	2	7	9		

The group name SAMPLE was defined on line 2 and referenced on lines 3,6,10 and 11. The group name TEST was defined on line 2 and referenced on lines 7 and 9.

STATEMENT AND FORMAT LABELS

```
STATEMENT LABELS
    0  4                                     4135  5          FMT
```

STATEMENT LABELS

Relative address of statement, followed by statement number. Inactive labels are printed with a zero address. A label becomes inactive when the compiler has deleted all references to it through optimization.

The following example contains the only reference to 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
```

Inactive labels and DO loop terminators are listed as INACTIVE with a zero address.

The type and activity are listed after the statement number:

Type	blank	Executable statement label
	FMT	Format label
	UNDEF	Label is undefined
Activity	blank	Label is active or referenced
	INACTIVE	Label is an inactive statement label
	NO REFS	Label is defined as a format label, and it is not referenced
	DEF LINE	Line number on which label appeared in source program
	REFERENCES	Line in which label was referenced

DO LOOP MAPS

This map, generated by the R = 2 or R = 3 option, is a printout of all DO loops in the source program and their properties. Loops are listed in order of appearance in the program.

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO 2 I = 1,M
    2 A(I) = A(I) + V
      RETURN
      END
```

```
      SUBROUTINE  SET

      SYMBOLIC REFERENCE MAP (R=3)

ENTRY POINTS     DEF LINE     REFERENCES
   23  INC           7          11
    2  SET           1

VARIABLES      SN  TYPE            RELOCATION
    0  A           REAL       ARRAY     F.P.        REFS        3        10    DEFINED        1        5        10
   44  I           INTEGER                          REFS        5      2*10    DEFINED        4        9
    0  M           INTEGER              F.P.        REFS        3         4          9   DEFINED        1
    0  V           REAL                 F.P.        REFS       10   DEFINED          1

STATEMENT LABELS              DEF LINE   REFERENCES
    0  1                          3           4
    0  2                         10           9

LOOPS  LABEL     INDEX      FROM-TO    LENGTH    PROPERTIES
   20  1         I            4 5        2B       INSTACK
   35  2         I            9 10       2B       INSTACK

STATISTICS
  PROGRAM LENGTH      57B       47
```

LOOPS	First word object program address in octal of beginning of loop
LABEL	Label associated with loop terminator. If none is present, it is an implied DO loop in an input/output list. The index of the DO follows. If preceded by an asterisk, the index is kept in memory during the loop.
FROM-TO	Initial and terminal line numbers of DO loop
LENGTH	Number of object program words generated for body of loop
PROPERTIES	If loop can be fully optimized, one of these messages is printed:
OPT	Loop has no properties which inhibit full optimization.
INSTACK	The instruction stack is a group of 8 (6000 series) or 12 (7600) 60-bit registers in the CPU computation section that holds program instruction words for execution. INSTACK means the loop is small enough to fit in this instruction stack and will usually run two to three times faster than loops that do not fit in the stack.

If loop is not fully 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 loop innermost in a nest.

COMMON BLOCKS

Common block symbols include common block names, and names declared in COMMON statements to be variables and arrays in common.

```
COMMON BLOCKS   LENGTH
      / /           12
```

COMMON BLOCKS	Block name
LENGTH	Total block length

When R = 1 or R = 2 is specified, only the above information is listed. When R = 3, the following details appear for each member declared in a COMMON statement.

MEMBERS	Relative address (distance from origin of common block)
BIAS NAME	Name of member of common block
(LENGTH)	Number of words allocated for member

If an equivalence class is linked to common, all members of the equivalence group become members of the common block. They are listed in the equivalence class printout.

```
COMMON BLOCKS   LENGTH     MEMBERS - BIAS NAME(LENGTH)
      / /           12                 0 X       (12)
```

EQUIVALENCE CLASSES

Equivalence symbols appear only when R = 3 is specified. All members of an equivalence group are listed. Any symbols added through linkage to common are not included.

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      COMMON A(1),B,C,D,  F,G,H
      INTEGER A,B,C,D,E(3,4),F,  H
      EQUIVALENCE (A,E,I)
      NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
1     A(J)=J

      WRITE (6,VLIST)
      STOP
      END
```

```
 EQUIV CLASSES   LENGTH   MEMBERS - BIAS NAME(LENGTH)
A      A             12              0 E       (12)        0 I    (1)
```

EQUIV	*ERROR* Class is in error (more than one member of an equivalence group is in common, or common block origin is extended by EQUIVALENCE statement or conflicting equivalencing attempted). BASE MEMBER — Class is in common blank — Other
CLASSES	If the equivalence group is not in common, the first member of the group (the member with the smallest object code address) is printed. If the group is in common, the name of the symbol in common which linked the equivalence group to the common block is printed. When an equivalence group is in common, the base member of the equivalence group is the first member of the common block.
MEMBERS	Equivalence group length
BIAS	Distance between an equivalence group member and the first member of the group. In an equivalence group A,B,C(10), C(1) is the base member, C(2) has a bias of 1, C(3) has a bias of 2, A and B have a bias of 9.
NAME	Name of equivalence group member
(LENGTH)	Number of words allocated for the member

Members of an equivalence group are printed in 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, program statistics are printed in octal and decimal.

```
STATISTICS
  PROGRAM LENGTH      145B       101
   BUFFER LENGTH     2033B      1051
    BLANK COMMON       14B        12
```

PROGRAM LENGTH	Program length including executable code, storage for variables not in common, constants, temporaries, etc., but excluding buffers and common blocks.
BUFFER LENGTH	Total space occupied by input/output buffers and FITs
COMMON LENGTH	Total common length, excluding blank common
BLANK COMMON	Length of blank common

```
PROGRAM     COME

                PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
                IMPLICIT INTEGER (A-F,H)
                DIMENSION E(3,4)
                COMMON A(1),B,C,D,  F,G,H
                EQUIVALENCE (A,E,I)
                NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

                DO 1 J = 1, 12
              1 A(J)=J

                WRITE (6,VLIST)
                STOP
                END
```

```
      PROGRAM    COME                                         CDC 6600 FTN V4.0-P271 OPT=1  09/03/71  06.39.08.      PAGE     2

      SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS
 2037   COME

VARIABLES      SN  TYPE          RELOCATION
    0  A          INTEGER    ARRAY    / /              1  B        INTEGER           / /
    2  C          INTEGER             / /              3  D        INTEGER           / /
    0  E          INTEGER    ARRAY    / /              4  F        INTEGER           / /
    5  G          REAL                / /              6  H        INTEGER           / /
    0  I          INTEGER             / /           2101  J        INTEGER

FILE NAMES         MODE
    0  OUTPUT                      0  TAPE6        FMT

NAMELISTS
       VLIST

STATEMENT LABELS
    0  1

COMMON BLOCKS    LENGTH
       / /         12

STATISTICS
  PROGRAM LENGTH      47B      39
   BUFFER LENGTH   2033B     1051
    BLANK COMMON      14B      12
```

DEBUGGING (USING REFERENCE MAP)

NEW PROGRAM

The reference map can be used to find incorrectly punched names as well as other items that will not appear as compilation errors. The basic technique consists of using the compiler as a verifier and correcting the fatal 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 (undeclared arrays)

Functions that should be in line instead of external

Variables or functions with incorrect type

Unreferenced format statements

Unused dummy arguments

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.

PROGRAM OUT

Program OUT illustrates the WRITE and PRINT statements.

Features:

- Control cards
- WRITE and PRINT statements
- Carriage control
- PROGRAM statement

```
PAT,T10,CM45000.
```

The job card must precede every job run under the SCOPE operating system. PAT is the job name. T10 specifies a maximum of 10 (octal) seconds central processor time, and CM45000 requests 45000 (octal) words of memory for the job.

```
FTN.
```

Specifies the FORTRAN Extended compiler and uses the default parameters. (section 11, part 1.)

```
LGO.
```

The binary object code is loaded and executed.

If no alternative files are specified on the FTN card, the FORTRAN Extended compiler reads from the file INPUT and outputs to two files: OUTPUT and LGO. Listings, diagnostics, and maps are output to OUTPUT and the relocatable object code to LGO.

```
7/8/9
```

The end-of-record card (EOR) or end-of-section card (EOS) separates control cards from the remainder of the INPUT file. The end-of-record card is a multipunch 7/8/9 in column 1; it must follow the SCOPE control cards in every job.

```
PROGRAM OUT (OUTPUT,TAPE6=OUTPUT)
```

The PROGRAM card identifies this as the main program with the name OUT and specifies the file OUTPUT. Output unit 6 will be referenced in the program. All files used by a program must be specified in the PROGRAM card of the main program.

TAPE6 = OUTPUT is included because output unit 6 is referenced in a WRITE statement. The unit number must be preceded by the letters TAPE. All data written to unit 6 will be placed in the SCOPE file OUTPUT and output to the printer.

```
WRITE (6,200) INK
```

The WRITE statement outputs the variable INK to output unit 6. If a PRINT statement had been used instead of WRITE:

```
   PRINT 200, INK
```

TAPE6 = OUTPUT would not be needed in the PROGRAM card; PROGRAM OUT (OUTPUT) would be sufficient.

```
100 FORMAT (*1 THIS WILL PRINT AT THE TOP OF A PAGE*)
```

This FORMAT statement uses * * to delimit the literal. 1 is a carriage control character which causes the line to be printed at the top of a page.

```
200 FORMAT (I5,* = INK OUTPUT BY WRITE STATEMENT*)
```

Although the variable INK is 4 digits, a specification of I5 is given because the first character is always interpreted as a control. In this case, the carriage control character is a blank and output will appear on the next line.

```
6/7/8/9
```

This is the end of file (EOF) or end of partition card; a multipunch 6/7/8/9 in column 1. This card must appear as the last card in each SCOPE job.

```
PAT,T10,CM45000.
FTN.
LGO.
7/8/9 in column 1
      PROGRAM OUT (OUTPUT,TAPE 6=OUTPUT)
      PRINT 100
 100  FORMAT (*1 THIS WILL PRINT AT THE TOP  OF A PAGE*)
      INK = 2000+4000
      WRITE (6,200) INK
  200 FORMAT (I5,* = INK OUTPUT BY WRITE STATEMENT*)
      PRINT 300, INK
  300 FORMAT (1H ,I4,30H = OUTPUT FROM PRINT STATEMENT)
      STOP
      END
6/7/8/9 in column 1
```

Output:

```
 THIS WILL PRINT AT THE TOP  OF A PAGE
6000 = INK OUTPUT BY WRITE STATEMENT
6000 = OUTPUT FROM PRINT STATEMENT
```

PROGRAM B

Program B generates a table of 64 characters indicating which character set is being used. The internal bit configuration of any character can be determined by its position in the table. Each character occupies six consecutive bits.

Features:

Octal constants

Simple DO loop

PRINT statement

FORMAT with H,/,I,X and A elements

The print statement PRINT1 has no input/output list; it prints out the heading at the top of the page using the information provided by the FORMAT statement on line 3. 25H specifies a Hollerith field of 25 characters, 1 is the carriage control character, and the two slashes // cause one line to be skipped before the next Hollerith field is printed. The slash at the end of the FORMAT specification skips another line before the program output is printed.

```
DO 3 I=1,8
J=I-1
```

These statements output numbers 0 through 7. A DO index cannot begin with a zero.

```
PRINT 2, J, NCHAR
```

Prints out 0 through 7 (the value of J) on the left and the 8 characters in NCHAR on the right. The first iteration of the DO loop prints NCHAR as it appears on line 4. The octal value 01 is a display code A, 02 is a B, 03 is a C, etc.

```
NCHAR=NCHAR+10 10 10 10 10 10 10 10 00 00B
```

The octal constant 10101010101010100000B is added to NCHAR; and when this is printed on the second iteration of the DO loop, the octal value 10 is printed as a display code H, 11 as I, 12 as J, etc. Compare these values with the Character Set in Section 1, Part 3.

```
BBBBB,T10,CM70000,P15.
MAP(OFF)
FTN.
LGO.
7/8/9 in column 1
      PROGRAM B (OUTPUT)
      PRINT 1
    1 FORMAT(25H1TABLE OF INTERNAL VALUES//12H     01234567,/)
      NCHAR= 00 01 02 03 04 05 06 07 00 00B
      DO 3 I = 1,8
      J=I-1
      PRINT 2, J,NCHAR
    2 FORMAT(I3,1X,A8)
3     NCHAR=NCHAR+10 10 10 10 10 10 10 10 00 00B
      STOP
      END
6/7/8/9 in column 1
```

Output:

```
  TABLE OF INTERNAL VALUES

     01234567

  0   ABCDEFG
  1  HIJKLMNO
  2  PQRSTUVW
  3  XYZ01234
  4  56789+-*
  5  /()$= ,.
  6  ≡[]%≠↱∨∧
  7  ↑↓<>≤≥¬;
```

PROGRAM MASK

Program MASK reads names and home states from data cards ignoring all but the first two letters of the state name. If the state name starts with the letters CA, the name is printed.

Feature:

Masking

```
1  FORMAT (1H1,5X,4HNAME,///)
   PRINT 1
```

The printer is directed to start a new page, print the heading NAME, and skip 3 lines.

```
3  READ 2,LNAME,FNAME,ISTATE,KSTOP
   IF(KSTOP.EQ.1)STOP
```

The last name is read into LNAME, first name into FNAME, and home state into ISTATE. The last card in the deck contains a one which will be read into KSTOP as a stop indicator. The IF statement on line 6 tests for the stop indicator.

```
 IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.777700000000000
K00000B))  GO TO 3
```

The relational operator .NE. tests to determine if the first two letters read from the data card into variable ISTATE match the two letters of the Hollerith constant CA. The last eight characters (48 bits) in ISTATE are masked and the two remaining characters are compared with the word containing the Hollerith constant CA, also similarly masked. If the bit string forming one word is not identical to the bit string forming the other word, ISTATE is not equal to CA and the IF statement test is true.

The bit configuration of CALIFORNIA, the Hollerith constant CA and the mask follows:

California

Hollerith	C	A	L	I	F	O	R	N	I	A
Octal	03	01	14	11	06	17	22	16	11	01
Bit	000011	000001	001100	001001	000110	001111	010010	001110	001001	000001

Constant CA

Hollerith	C	A	blank	blank	blank	blank	blank	blank	blank	blank
Octal	03	01	55	55	55	55	55	55	55	55
Bit	000011	000001	101101	101101	101101	101101	101101	101101	101101	101101

Mask

Octal	77	77	00	00	00	00	00	00	00	00
Bit	111111	111111	000000	000000	000000	000000	000000	000000	000000	000000

When the masking expression (ISTATE.AND.77770000000000000000B) is completed, the first two characters of CALIFORNIA remain the same and last eight characters are zeroed out. The AND operation follows:

```
000011  000001  001100  001001  000110  001111  010010  001110  001001  000001

111111  111111  000000  000000  000000  000000  000000  000000  000000  000000
------------------------------------------------------------------------------
000011  000001  000000  000000  000000  000000  000000  000000  000000  000000
```

When (2HCA.AND.77770000000000000000B) is evaluated, the same result is obtained. Thus, in both words, all bits but those forming the first two characters will be masked, making a valid basis for comparing the first two characters of both words. If the result of the mask is true, the last name and first name are printed (statement 10), otherwise the next card is read.

```
      PROGRAM MASK (INPUT,OUTPUT)
1     FORMAT (1H1,5X,4HNAME,///)
      PRINT 1
2     FORMAT (3A10,I1)
3     READ 2,LNAME,FNAME,ISTATE,KSTOP
      IF(KSTOP.EQ.1)STOP
C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD
      IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.7777000000000000
     K0000B))  GO TO 3
11    FORMAT(5X,2A10)
10    PRINT 11,LNAME,FNAME
      GO TO 3
      END
```

Data cards:

```
BROWN,    PHILLIP M.CA
BICARDI,  R. J.     KENTUCKY
CROWN,    SYLVIA    CAL
HIGENBERF,ZELDA     MAINE
MUNCH,    GARY G.   CALIF.
SMITH     SIMON     CA
DEAN      ROGER     GEORGIA
RIPPLE    SALLY     NEW YORK
JONES     STAN      OREGON
HEATH     BILL      NEW YORK
                              1
```

Output:

```
NAME



BROWN,    PHILLIP M.
CROWN,    SYLVIA
MUNCH,    GARY G.
SMITH     SIMON
```

PROGRAM EQUIV

Program EQUIV places values in variables that have been equivalenced and prints these values using the NAMELIST statement.

Features:

EQUIVALENCE statement

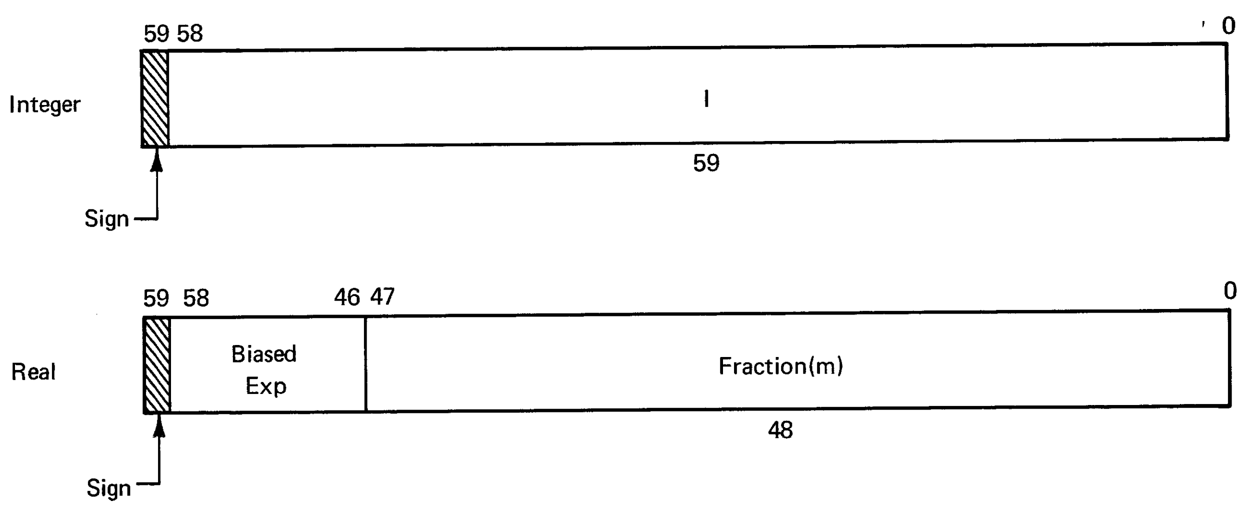
NAMELIST statement

```
EQUIVALENCE (X,Y),(Z,I)
```

Two real variables X and Y are equivalenced; the two variables share the same location in storage, which can be referred to as either X or Y. Any change made to one variable changes the value of the others in an equivalence group as illustrated by the output of the WRITE statement, in which both X and Y have the value 2. The storage location shared by X and Y contained first 1. (X=1.) then 2. (Y=2.).

The real variable Z and the integer variable I are equivalenced, and the same location can be referred to as either real or integer. Since integer and real internal formats differ, however, the output values will not be the same.

For example, the storage location shared by Z and I contained first 3. then the integer value 4 When I is output, no problem arises; an integer value is referred to by an integer variable name. However, when this same integer value is referred to by a real variable name, the value 0.0 is output. The internal format of real and integer values differ.



Although they can be referred to by names of different types, the internal bit configuration does not change. An integer value output as a real variable does not have an exponent and its value will be small.

When variables of different types are equivalenced, the value in the storage location must agree with the type of the variable name; or unexpected results may be obtained.

```
WRITE(6,OUTPUT)
```

This NAMELIST WRITE statement outputs both the name and the value of each member of the NAMELIST group OUTPUT defined in the statement NAMELIST/OUTPUT/X,Y,Z,I. The NAMELIST group is preceded by the group name, OUTPUT, and terminated by the characters $END.

```
PROGRAM EQUIV (OUTPUT,TAPE6=OUTPUT)
EQUIVALENCE (X,Y),(Z,I)
NAMELIST/OUTPUT/X,Y,Z,I
X=1.
Y=2.
Z=3.
I=4
WRITE(6,OUTPUT)
STOP
END
```

Output:

```
$OUTPUT

X        =  0.2E+01,

Y        =  0.2E+01,

Z        =  0.0,

I        =  4,

$END
```

PROGRAM COME

Program COME places variables and arrays in common and declares another variable and array equivalent to the first element in common. It places the numbers 1 through 12 in each element of the array A and outputs values in common using the NAMELIST statement.

Features:

COMMON and EQUIVALENCE statements

NAMELIST statement

```
COMMON A(1),B,C,D, F,G,H
```

Variables are stored in common in the order of appearance in the COMMON statement A(1),B,C,D,F,G,H. Variables can be dimensioned in the COMMON statement; and in this instance, A is dimensioned so that it can be subscripted later in the program. If A were not dimensioned, it could not be used as an array in statement 1.

```
INTEGER A,B,C,D,E(3,4),F,H
```

All variables with the exception of G are declared integer. G is implicitly typed real.

```
EQUIVALENCE(A,E,I)
```

The EQUIVALENCE statement assigns the first element of the arrays A and E and an integer variable I to the same storage location. Since A is in common, E and I will be in common. Variables and array elements are assigned storage as follows:

Relative Address	0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
	I											
	E(1,1)	E(2,1)	E(3,1)	E(1,2)	E(2,2)	E(3,2)	E(1,3)	E(2,3)	E(3,3)	E(1,4)	E(2,4)	E(3,4)
	A(1)	B	C	D	F	G	H					
		A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)	A(11)	A(12)

```
    DO 1 J=1,12
1   A(J)=J
```

The DO loop places values 1 through 12 in array A. The first element of array A shares the same storage location with the first element of array E. Since B is equivalent to E(2,1), A(2) is equivalent to B, A(3) to C, A(4) to D, etc.

Any change made to one member of an equivalence group changes the value of all members of the group. When 1 is stored in A, both E(1,1) and I have the value 1. When 2 is stored in A(2), B and E(2,1) have the value 2. Although B and E(2,1) are not explicitly equivalenced to A(2), equivalence is implied by their position in common.

The implied equivalence between the array elements and variables is illustrated by the output.

The NAMELIST statement is used for output. A NAMELIST group, V, containing the variables and arrays A,B,C,D,E,F,G,H,I is defined. The NAMELIST WRITE statement, WRITE(6,V), outputs all the members of the group in the order of appearance in the NAMELIST statement. Array E is output on one line in the order in which it is stored in memory. There is no indication of the number of rows and columns (3,4).

G is equivalent to E(3,2) and yet the output for E(3,2) is 6 and G 0.0. G is type real and E is type integer. When two names of different types are used for the same element, their values will differ because the internal bit configuration for type real and type integer differ (refer to Program EQUIV).

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      COMMON A(1),B,C,D,  F,G,H
      INTEGER A,B,C,D,E(3,4),F,  H
      EQUIVALENCE (A,E,I)
      NAMELIST/V/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,V)
      STOP
      END
```

Output:

```
$V
A         =  1,
B         =  2,
C         =  3,
D         =  4,
E         =  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,  11,  12,
F         =  5,
G         =  0.0,
H         =  7,
I         =  1,
$END
```

PROGRAM LIBS

Program LIBS illustrates library subroutines provided by FORTRAN Extended.

Features:

EXTERNAL used to pass a library subroutine name as a parameter to another library routine.

Division by zero.

LEGVAR used to test for overflow or divide error conditions.

Library functions used:

LOCF

LEGVAR

Library subroutines used:

DATE

TIME

SECOND

RANGET

DATE is a library subroutine which returns the date entered by the operator from the console. DATE is declared external because it is used as a parameter to the function LOCF. Declaring DATE external does not prevent its use as a library subroutine in this program.

```
   PRINT2,TODAY,CLOCK
2  FORMAT(*1TODAY=*Y, A10, * CLOCK=*,A10)
```

These statements print the date and time. The leading and trailing blanks appear with the 10 alphanumeric characters returned by the subroutine DATE because the operator typed in the date this way. However, since he may choose to use a 4-digit year, it may be prudent to use A11 in the output FORMAT specification to guarantee at least one leading space. The value returned by TIME is changed by the system once a second, and the position of the digits remain fixed; a leading blank always will appear. The format of DATE and TIME can be checked by observing any listing, as the routines DATE and TIME are used by the compiler to print out the date and time at the top of compiler output listings.

```
CALL SECOND(TYME)
```

When SECOND is called, the variable name TYME is used. A variable name cannot be spelled the same as a program unit name. If Program LIBS had not called the subroutine TIME, a variable name could be spelled TIME.

```
LOCATN=LOCF(DATE)
```

DATE is not a variable name as it appears in an EXTERNAL statement.

Library function LOCF returns the address of DATE.

```
CALL RANGET(SEED)
```

Library subroutine RANGET returns the seed used by the random number generator RANF if it is called. If RANGET is called after RANF has been used, RANGET will return the value currently being processed by the random number generator. With the library subroutine RANSET, this same value could be used to initialize the random number generator at a later date.

```
   PRINT3, TYME, LOCATN, LOCATN, SEED, SEED
3  FORMAT(*OTHE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*//* LOCATION OF
  1 DATE ROUTINE IS=*,O15,* OR*,I7,* IN DECIMAL.*/*OTHE INITIAL VALUE
  2 OF THE RANF SEED IS *,O22,*, OR*,G30.15,* IN G30.15 FORMAT.*)
```

These statements print out the values returned by the routines SECOND, LOCF, and RANGET.

Asterisks are used to delineate Hollerith fields in the format specification to illustrate the point that excessive use of asterisks can be extremely difficult to follow.

```
Y=0.0
WOW=7.2/Y
IF(0.NE. LEGVAR(WOW))PRINT4,WOW
```

These statements illustrate the use of the library function LEGVAR within an IF statement to test the validity of division by zero. LEGVAR checks the variable WOW. This function returns a result of -1 if the variable is indefinite, +1 if it is out of range, and 0 if it is normal. Comparing the value returned by LEGVAR with 0 shows that the number is either indefinite or out of range. The output RRR shows the variable is out of range. Division by zero is allowed on the 6000/7000 and CYBER 70 series computers and there is a representation for an infinite value (refer to section 4, part 3).

The line of -*-* on the output is produced by the FORMAT specification in statement number 4 50(2H*-).

```
      PROGRAM LIBS (OUTPUT)
C
      EXTERNAL DATE
C
      CALL DATE (TODAY)
      CALL TIME (CLOCK)

      PRINT 2, TODAY, CLOCK
    2 FORMAT(*1TODAY=*, A10, * CLOCK=*, A10)
C
      CALL SECOND(TYME)
      LOCATN=LOCF(DATE)
      CALL RANGET(SEED)

      PRINT 3,TYME, LOCATN, LOCATN, SEED, SEED
    3 FORMAT(*0THE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*//* LOCATION OF
     1 DATE ROUTINE IS=*,O15,* OR*,I7,* IN DECIMAL.*/*0THE INITIAL VALUE
     2 OF THE RANF SEED IS*,O22,*, OR*,G30.15,* IN G30.15 FORMAT.*)
C
      Y=0.0
      WOW=7.2/Y
      IF(0 .NE. LEGVAR(WOW))PRINT4,WOW
      STOP
    4 FORMAT(1H0,50(2H*-)/* DIVIDE ERROR, WOW PRINTS AS=*,G10.2)
      END
```

```
TODAY= 08/27/71  CLOCK= 18.28.21.

THE ELAPSED CPU TIME IS     .29800      SECONDS.

LOCATION OF DATE ROUTINE IS=000000000006410 OR   3336 IN DECIMAL.

THE INITIAL VALUE OF THE RANF SEED IS  17171274321477413155, OR         .170998394044023     IN G30.15 FORMAT.

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
DIVIDE ERROR, WOW PRINTS AS=     RRRRR
```

PROGRAM PIE

Program PIE calculates an approximation of the value of π .

Feature:

Library function RANF

The random number generator, RANF, is called twice during each iteration of the DO loop, and the values obtained are stored in the variables X and Y.

```
DATA CIRCLE,DUD/2*0.0/
```

The DATA statement initializes the variables CIRCLE and DUD with the value 0.0.

Each time the DO loop is iterated, a random number, uniformly distributed over the range 0-1, is returned by the library function RANF, and this value is stored in the variable X. The value of X will be $0 \leq X < 1$. DUD is a dummy argument which must be used when RANF is called.

```
Y=RANF(DUD)
```

RANF is referenced again; this time to obtain a value for Y.

```
IF(X*X*Y*Y.LE.1.)CIRCLE=CIRCLE+1.
```

The IF statement and the arithmetic expression 4.*CIRCLE/10000. calculate an approximation of the value of π. The value of π is calculated using Monte Carlo techniques. The IF statement counts those points whose distance from CIRCLE(0.0) is less than one. The ratio of the number of points within the quarter circle to the total number of points approximates 1/4 of π. The value PI is printed by the NAMELIST statement WRITE(6,OUT)

```
      PROGRAM PIE(OUTPUT,TAPE6=OUTPUT)
      DATA CIRCLE,DUD/2*0.0/
      NAMELIST/OUT/PI

      DO 1 I = 1,10000
      X=RANF(DUD)
      Y=RANF(DUD)
      IF(X*X+Y*Y.LE.1.)CIRCLE=CIRCLE+1.
1     CONTINUE

      PI=4.*CIRCLE/10000.
      WRITE(6,OUT)

      STOP
      END
```

Output:

```
      $OUT

      PI        =  0.31596E+01,

      $END
```

PROGRAM ADD

Program ADD illustrates the use of the DECODE statement.

Features:

DECODE statement.

ENCODE and DECODE

ENCODE and DECODE are simpler to understand when related to the WRITE and READ statements.

DECODE (READ)

A READ statement places the image of each card read into an input buffer. The card image occupies eight computer words, each word containing ten display code characters. Compiler routines convert the character string in the card image into floating point, integer or logical values, as specified by the FORMAT statement, and store these values into the locations associated with the variables named in the list.

With DECODE, the information in the input buffer comes from the array specified in the DECODE statement. The number of words moved to the input buffer from the array is determined by the record length. Since the input buffer is 150 words long, the maximum record length is 150.

With the READ statement, when the FORMAT specification indicates a new record is to be processed (by a / or the final right parenthesis of the FORMAT statement) a new record is obtained by reading another card into the input buffer.

With the DECODE statement, when the FORMAT statement indicates a new record is to be processed (/ or final right parenthesis), as many words as indicated by the record length are obtained from the array and placed in the input buffer.

ENCODE (WRITE)

A WRITE statement causes the output buffer to be cleared to 150 spaces. Data in the WRITE statement list is converted by compiler routines into a character string according to the format specified in the FORMAT statement, and placed in the output buffer. When the FORMAT statement indicates an end of a record with either a / or the final right parenthesis, the character string is passed from the output buffer to the SCOPE output system; the output buffer area is reset to spaces, and the next string of characters is placed in the buffer.

The ENCODE statement is processed by compiler routines in the same way as the WRITE statement; but when a record is output, the character string is placed into the array specified within the parentheses of the ENCODE statement. The number of words moved from the output buffer to the array is determined by the record length.

The number of computer words in each ENCODE or DECODE record is determined by dividing the record length by 10 and rounding up. For example, a record length of 33 requires 4 words, and a record length of 71 requires 8 words.

As a mnemonic aid, it may be useful to remember READ ends with a D and corresponds to DECODE, WRITE ends with an E and corresponds to ENCODE.

In the following program, the format of data on card input is unknown, but column 1 is a key to the type of data. The complete record is read under A format, and a test is made on column 1 to determine the type of data. If column 1 contains a 2, a set of numbers is to be summed and printed out. Using the DECODE statement, these numbers are converted from A format to I format and written in the list variables, INK. The values in the array INK are added and stored in ITOT.

```
      PROGRAM ADD (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      DIMENSION CARD (8), INK (77)
    2 READ (5,100) KEY1,CARD
  100 FORMAT (I1,7A10,A9)
      IF (EOF(5)) 80,90
   90 IF (KEY1-2) 3,8,3
    3 CALL ERROR1
      GO TO 2
    8 WRITE (6,300) CARD
  300 FORMAT (1H1,7A10,A7///)
      DECODE (77,17,CARD) INK
   17 FORMAT (77I1)
      ITOT = 0
      DO 4 I = 1,77
    4 ITOT = ITOT + INK(I)
      ISAVE = ITOT
      WRITE (6,200) ISAVE
  200 FORMAT (19X,*TOTAL OF 77 SCORES ON CARD = *,I10)
  80  STOP
      END


      SUBROUTINE ERROR1
      WRITE (6,1)
    1 FORMAT (5X,*NUMBER IS NOT 2*)
      RETURN
      END
```

Output:

```
2353689963214521354687963214568796321453698745123695821214525632145879632l4

                    TCTAL OF 77 SCORES ON CARD =          342
```

```
DIMENSION CARD(8),INK(77)
```

CARD is dimensioned 8 to receive card input.

	Card(1)	Card(2)	Card(3)	Card(4)	Card(5)	Card(6)	Card(7)	Card(8)
2	3536899632	1452135468	7963214568	7963214536	9874512369	5821214525	6321458794	321400000

INK is dimensioned 77. As each character from the card is decoded, it will become an element of the array INK.

```
  2  READ(5,100)KEY1,CARD
100  FORMAT(I1,7A10,A9)
```

The first column of the card, which is known to be integer, is read into KEY1 under I format. The remaining 77 characters and 2 blanks are read into the array CARD under A format, as the type of input is unknown.

```
IF (EOF(5)) 80,90
```

A test is made for end of file. If the last card has been read, the program terminates. If not, the first digit on the card is tested for 2 (statement 90); and if it is 2, the array CARD is printed (statement 8). An error routine is called if the first digit on the card is not 2.

```
DECODE (77,17,CARD) INK
```

The data on the input card is decoded. 77 characters in the array CARD are read into the array INK under I format; they are converted from alphanumeric format to integer format.

```
   DO 4 I=1,77
4  ITOT=ITOT+INK(I)
```

The DO loop totals the elements of the array INK (the 77 numbers on the card).

```
     WRITE(6,200)ISAVE
200  FORMAT(19X,*TOTAL OF 17 SCORES ON CARD = *,I10)
```

Prints the total and a heading.

PROGRAM PASCAL

Program PASCAL produces a table of binary coefficients (Pascal's triangle).

Features:

- Nested DO loops
- DATA statement
- Implied DO loop

```
INTEGER L(11)
```

L is defined as an 11-element integer array.

```
DATA L(11)/1/
```

The DATA statement stores the value 1 in the last element of the array L. When the program is executed L(11) has the initial value 1.

```
PRINT 4,(I,I=1,11)
```

This statement prints the headings. The implied DO loop generates the values 1 through 11 for the column headings.

```
PRINT 3,(L(J),J=K,11)
```

This is a more complicated example of an implied DO loop. The index value J is used as a subscript instead of being printed. The end of the array is printed from a variable starting position. The 1, which appears on the diagonal in the output is not moving in the array; it is always in L(11); but the starting point is moving.

```
DO 2 I=1,10
K=11-1
```

These statements illustrate the technique of going backwards through an array. As I goes from 1 to 10, K goes from 10 to 1. The increment value in a DO statement must be positive, therefore,

```
DO 2 I=1,10
K=11-I
```

provides a legal method of writing the illegal statement DO 2 K=10,1,-1.

```
   DO 1 J=K,10
1  L(J)=L(J)+L(J+1)
```

This inner DO loop generates the line of values output by statement number 2. When control reaches statement 2, the variable J can be used again because statement number 2 is outside the inner DO loop. However, if I were used in statement 2 instead of J, the statement 2 PRINT 3,(L(I),I=K,11) would be an error. Statement 2 is inside the inner DO loop and would change the value of the index from within the DO loop. Changing the value of a DO index from inside the loop is illegal and will cause a fatal error or a never ending loop.

```
      PROGRAM PASCAL (OUTPUT)
      INTEGER L(11)
      DATA L(11) /1/
C
      PRINT4, (I,I=1,11)
4     FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/
     $11I5)
      DO 2 I = 1,10
      K=11-I
      L(K)=1
      DO 1 J = K,10
1      L(J)=L(J)+L(J+1)
2     PRINT3,(L(J),J=K,11)
3     FORMAT(11I5)

      STOP
      END
```

PROGRAM X

Program X references a function EXTRAC which squares the number passed as an argument.

Features:

- Referencing a function
- Function type

Program X illustrates that a function type must agree with the type associated with the function name in the calling program.

```
K=EXTRAC(7)
```

Since the first letter of the function name EXTRAC is E, the function is implicitly typed real. EXTRAC is referenced, and the value 7 is passed to the function as an argument. However, the function subprogram is explicitly defined integer, INTEGER FUNCTION EXTRAC(K), and the conflicting types produce erroneous results.

The argument 7 is integer which agrees with the type of the dummy argument K in the subprogram. The result 49 is correctly computed. However, when this value is returned to the calling program, the integer value 49 is returned to the real name EXTRAC; and an integer value in a real variable produces an erroneous result (refer to program EQUIV).

This problem arises because the programmer and the compiler regard a program from different viewpoints. The programmer often considers his complete program to be one unit whereas the compiler treats each program unit separately. To the programmer, the statement

```
INTEGER FUNCTION EXTRAC(K)
```

defines the function EXTRAC integer. The compiler, however, compiles integer function EXTRAC and the main program separately. In the subprogram EXTRAC is defined integer, in the main program it is defined real. Information which the main program needs regarding a subprogram must be supplied in the main program - in this instance the type of the function.

There is no way for the compiler to determine if the type of a program unit agrees with the type of the name in the calling program; therefore, no diagnostic help can be given for errors of this kind.

```
      PROGRAM X (OUTPUT)
C     WITH EXTRAC DECLARED INTEGER THE RESULT SHOULD BE 49, OTHERWISE IT
C            WILL BE ZERO
      K = EXTRAC(7)
      PRINT 1, K
    1 FORMAT (1H1,I5)
      STOP
      END



      INTEGER FUNCTION EXTRAC (K)
      EXTRAC = K*K
      RETURN
      END
```

Output:

```
    0
```

PROGRAM VARDIM

Program VARDIM illustrates the use of variable dimensions to allow a subroutine to operate on arrays of differing size.

Features:

Passing an array to a subroutine as a parameter.

A subroutine name used as a parameter passes the address of the beginning of the array and no dimension information.

```
COMMON X(4,3)
```

Array X is dimensioned (4,3) and placed in common.

```
REAL Y(6)
```

Array Y dimensioned (6) is explicitly typed real. It is not in common.

```
CALL IOTA(X,12)
```

The subroutine IOTA is called. The first parameter to IOTA is array X, and the second parameter is the number of elements in that array, 12. The number of elements in the array rather than the dimensions (4,3) is used which is legal.

```
SUBROUTINE IOTA(A,M)
DIMENSION A(M)
```

Subroutine IOTA has variable dimensions. Array A is given the dimension M. Whenever the main program calls IOTA, it can provide the name and the dimensions of the array; since A and M are dummy arguments, IOTA can be called repeatedly with different dimensions replacing M at each call.

```
CALL IOTA(X,12)
```

When IOTA is called by the main program, the actual argument X replaces A; and 12 replaces M.

```
   DO 1 I=1,M
1  A(I)=I
```

The DO loop places the numbers 1 through 12 in consecutive elements of array X.

```
CALL IOTA(Y,6)
```

When IOTA is called again, Y replaces A and 6 replaces M; and numbers 1 through 6 are placed in consecutive elements of array Y. Notice the type of the arguments in the calling program agree with the type of the arguments in the subroutine. X and A are real, 12 and M are integer.

Names used in the subroutine are related to those in the calling program only by their position as arguments. If a variable I was in the calling program, it would be completely independent of the variable I in the subroutine IOTA.

The WRITE statement outputs the arrays X and Y.

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA(X,12)
      CALL IOTA(Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (*1ARRAY X = *,12F6.0,5X,*ARRAY Y = *6F6.0)
      STOP
      END
      SUBROUTINE IOTA (A,M)
C     IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
C     STARTING AT 1
      DIMENSION A(M)
      DO 1 I = 1,M
1     A(I)=I
      RETURN
      END
```

```
ARRAY X =     1.    2.    3.    4.    5.    6.    7.    8.    9.   10.   11.   12.     ARRAY Y =     1.    2.    3.    4.    5.    6.
```

PROGRAM VARDIM2

VARDIM2 is an extension of program VARDIM. Subroutine IOTA is used; in addition, another subroutine and two functions are used.

Features:

- Multiple entry points
- Variable dimensions
- EXTERNAL statement
- COMMON used for communication between program units
- Passing values through COMMON
- Use of library functions ABS and FLOAT
- Calling functions through several levels
- Passing a subprogram name as an argument

Program VARDIM2 describes the method of a main program calling subprograms and subprograms calling each other. Since the program is necessarily complex, each subprogram is described separately followed by a description of the main program.

SUBROUTINE IOTA

SUBROUTINE IOTA is described in program VARDIM.

SUBROUTINE SET

SUBROUTINE SET(A,M,V) places the value V into every element of the array A. The dimension of A is specified by M.

Subroutine SET has an alternate entry point INC. When SET is entered at ENTRY INC, the value V is added to each element of the array A. The dimension of A is specified by M.

The DO loop in subroutine SET clears the array to zero.

FUNCTION AVG

This function computes the average of the first J elements of common. J is a value passed by the main program through the function PVAL.

This function subprogram is an example of a main program and a subprogram sharing values in common. The main program declares common to be 12 words and FUNCTION AVG declares common to be 100 words. Function AVG and the main program share the first 12 words in common. Values placed in common by the main program are available to the function subprogram.

The number of values to be averaged is passed to FUNCTION PVAL by the statement AA = PVAL(12,AVG) and function PVAL passes this number to function AVG: PVAL = ABS(WAY(SIZE))

```
COMMON A(100)
```

Function AVG declares common 100 so that varying lengths (less than 100) can be used in calls. In this instance, only 12 of the 100 words are used.

```
   DO 1 I=1,J
1  AVG=AVG+A(I)
```

The DO loop adds the 12 elements in common.

```
AVG=AVG/FLOAT(J)
```

This statement finds the average. The library function FLOAT is used to convert the integer 12 to a floating point (real) number to avoid mixed mode arithmetic.

The average is returned to the statement PVAL = ABS(WAY(SIZE)) in function PVAL.

FUNCTION PVAL

Function PVAL references a function specified by the calling program to return a value to the calling program. This value is forced to be positive by the library function ABS.

The main program first calls PVAL with the statement AA=PVAL(12,AVG), passing the integer value 12 and the function AVG as parameters.

```
INTEGER SIZE
```

PVAL declares SIZE integer - the type of the argument in the main program (integer 12) agrees with the corresponding dummy argument (SIZE) in the subprogram.

```
PVAL=ABS(WAY(SIZE))
```

The value of PVAL is computed. This value will be returned to the main program through the function name PVAL. Two functions are referenced by this statement; the library function ABS and the user written function AVG. The actual arguments 12 and AVG replace SIZE and WAY.

```
PVAL=ABS(AVG(12))
```

Function AVG is called, and J is given the value 12. The average of the first 12 elements of common are computed by AVG and returned to function PVAL. Library function ABS finds the absolute value of the value returned by AVG.

```
AM=PVAL(12,MULT)
```

In this statement in the main program, PVAL is referenced again. This time the function MULT replaces WAY.

FUNCTION MULT

MULT multiplies the first and twelth words in COMMON and subtracts the product from the average (computed by the function AVG) of the first J/2 words in common.

```
COMMON ARRAY(12)
```

Common is declared 12; MULT shares the first 12 words of common with the main program.

```
MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
```

The twelth and first element in common are multiplied and the average of J/2 is subtracted. This is an example of a subprogram calling another subprogram - the function AVG is used to compute the average.

MAIN PROGRAM — VARDIM2

The main program calls the subroutines and functions described.

```
COMMON X(4,3)
```

Twelve elements in the array X are declared to be in common.

```
REAL Y(6)
```

The real array Y is dimensioned 6.

```
EXTERNAL MULT, AVG
```

Function names MULT and AVG are declared EXTERNAL. Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program. Otherwise it would be treated by the compiler as a variable name.

```
CALL SET(Y,6,0.)
```

Subroutine SET is called. The arguments (Y,6,0.) replace the dummy arguments (A,M,V).

```
   DIMENSION Y (6)
   DO 1 I = 1,6
1  Y(I) = 0.0
```

The array Y is set to zero. The NAMELIST output shows the 6 elements of Y contain zero.

```
CALL IOTA(X,12)
```

Subroutine IOTA is called. X and 12 replace the dummy arguments A and M

```
   DIMENSION X (12)
   DO 1 I=1,12
1  X(I) = I
```

the value of the subscript is placed in each element of the array X. Program VARDIM output shows the value of X is 1 through 12.

```
CALL INC(X,12,-5.)
```

Subroutine SET is called, this time through entry point INC. The arguments (X,12,-5.) replace the dummy arguments (A,M,V)

```
   DO 2 I=1,12
2  X(I) = X(I) + -5.
```

-5. is added to each element in the array X. Program VARDIM2 output shows X is now -4,-3,-2, -1,0,1,2,3,4,5,6,7

```
AA=PVAL(12,AVG)
```

Function PVAL is called and its value replaces AA.

```
AM=PVAL(12,MULT)
```

Function PVAL is called again with different arguments and the value replaces AM.

```
      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
C     THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBPROGRAM CONCEPTS
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      NAMELIST/V/X,Y,AA,AM
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
      AA=PVAL(12,AVG)
      AM=PVAL(12,MULT)
      WRITE(6,V)
      STOP
      END
```

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO 2 I = 1,M
    2 A(I) = A(I) + V
      RETURN
      END
```

```
      SUBROUTINE IOTA (A,M)
C     IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C          THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=I
      RETURN
      END
```

```
      FUNCTION PVAL(SIZE,WAY)
C  PVAL COMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALUE IS RETURNED
C     BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED.  SIZE IS AN INTEGER
C     VALUE PASSED ON TO THE FUNCTION.
      INTEGER SIZE
      PVAL=ABS(WAY(SIZE))
      RETURN
      END
```

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
      COMMON A(100)
      AVG=0.
      DO 1 I = 1,J
    1 AVG=AVG+A(I)
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
      REAL FUNCTION MULT(J)
C      MULT MULTIPLIES THE FIRST AND TWELTH ELEMENTS OF COMMON AND
C      SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C     BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
C
      COMMON ARRAY(12)
      MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
      RETURN
      E  N  D
```

```
SV

X      = -0.4E+01, -0.3E+01, -0.2E+01, -0.1E+01,  0.0,  0.1E+01,  0.2E+01,      0.3E+01,  0.4E+01,  0.5E+01,
  0.6E+01,  0.7E+01,

Y      =  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,

AA     =  0.15E+01,

AM     =  0.265E+02,

$END
```

PROGRAM CIRCLE

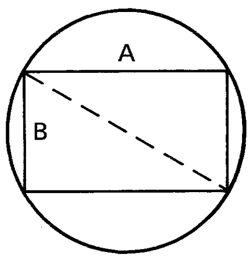
Program CIRCLE finds the area of a circle which circumscribes a rectangle.

Features:

Definition and use of both FUNCTION subprograms and statement functions.

This program has a hidden bug. We suggest you read the text from the start if you intend to find it.

A programmer wrote the following program to find the area of a circle which circumscribes a rectangle, and wrote a function named DIM to compute the diameter of the circle.



The area of a circle is πR^2, which is approximately the same as 3.1416/4*Diameter**2.

```
      PROGRAM CIRCLE (OUTPUT)
      A=4.0
      B=3.0
      AREA=3.1416/4.0*DIM(A,B)**2
      PRINT 1, AREA
1     FORMAT(G20.10)
      STOP
      END
      FUNCTION DIM(X,Y)
      DIM=SQRT(X*X+Y*Y)
      RETURN
      END
```

Output:

```
      .7854000000
```

The programmer was completely baffled by the result; he felt the area of a circle circumscribing a rectangle 12 square inches should be more than .785! He consulted another programmer who quickly pointed out that a simple function like DIM should have been written as a statement function. Since FORTRAN Extended compiles statement functions inline, it would execute much faster because no jump nor return jump would be generated by the function.

The programmer rewrote his program as follows:

```
      PROGRAM CIRCLE (OUTPUT)
      DIM(X,Y)=SQRT(X*X+Y*Y)
      A=4.0
      B=3.0
      AREA=3.1416/4.0*DIM(A,B)**2
      PRINT 1, AREA
    1 FORMAT (G20.10)
      STOP
      END
```

and obtained the correct result.

When the programmer wrote his function subprogram, he used the same name as a library intrinsic function. If the name of an intrinsic function is used for a user written function, the user written function is ignored.

SOURCE PROGRAM CHARACTER SETS III-1

Since the character set is selected when FORTRAN Extended is installed, the user should check with his installation to determine which character set is being used.

Installation options allow the user to select an 026 or an 029 keypunch, or to override this selection by punching a 26 or 29 in columns 79 and 80 of the SCOPE job card, or any 7/8/9 end-of-record card. The keypunched 26 or 29 remains in effect for the remainder of the job or until it is reset by a different mode selection on another 7/8/9 card.

ASCII 64-CHARACTER SUBSET*

Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code	Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code
00	: †	8-2	8-2	072	40	5	5	5	065
01	A	12-1	12-1	101	41	6	6	6	066
02	B	12-2	12-2	102	42	7	7	7	067
03	C	12-3	12-3	103	43	8	8	8	070
04	D	12-4	12-4	104	44	9	9	9	071
05	E	12-5	12-5	105	45	+	12	12-8-6	053
06	F	12-6	12-6	106	46	–	11	11	055
07	G	12-7	12-7	107	47	*	11-8-4	11-8-4	052
10	H	12-8	12-8	110	50	/	0-1	0-1	057
11	I	12-9	12-9	111	51	(	0-8-4	12-8-5	050
12	J	11-1	11-1	112	52	)	12-8-4	11-8-5	051
13	K	11-2	11-2	113	53	$	11-8-3	11-8-3	044
14	L	11-3	11-3	114	54	=	8-3	8-6	075
15	M	11-4	11-4	115	55	blank	no punch	no punch	040
16	N	11-5	11-5	116	56	, (comma)	0-8-3	0-8-3	054
17	O	11-6	11-6	117	57	. (period)	12-8-3	12-8-3	056
20	P	11-7	11-7	120	60	#	0-8-6	8-3	043
21	Q	11-8	11-8	121	61	' (apostrophe)	8-7	8-5	047
22	R	11-9	11-9	122	62	!	0-8-2	12-8-7	041
23	S	0-2	0-2	123	63	%	8-6	0-8-4	045
24	T	0-3	0-3	124	64	" (quote)	8-4	8-7	042
25	U	0-4	0-4	125	65	_ (underline)	0-8-5	0-8-5	137
26	V	0-5	0-5	126	66	]	11-0 or 11-8-2	11-0 or 11-8-2	135
27	W	0-6	0-6	127	67	&	0-8-7	12	046
30	X	0-7	0-7	130	70	@	11-8-5	8-4	100
31	Y	0-8	0-8	131	71	?	11-8-6	0-8-7	077
32	Z	0-9	0-9	132	72	[	12-0 or 12-8-2	12-0 or 12-8-2	133
33	0	0	0	060	73	>	11-8-7	0-8-6	076
34	1	1	1	061	74	<	8-5	12-8-4	074
35	2	2	2	062	75	\	12-8-5	0-8-2	134
36	3	3	3	063	76	^ (circumflex)	12-8-6	11-8-7	136
37	4	4	4	064	77	; (semicolon)	12-8-7	11-8-6	073

†This character is lost on even parity magnetic tape.

*BCD representation is used when data is recorded on even parity magnetic tape. In this case, the octal BCD/display code correspondence is the same as for the CDC 64-character set.

CDC 64-CHARACTER SET

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	: †	8-2	8-2	00*	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	−	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	%	8-6	0-8-4	16
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	∨	11-0 or 11-8-2	11-0 or 11-8-2	52
27	W	0-6	0-6	26	67	∧	0-8-7	12	37
30	X	0-7	0-7	27	70	↑	11-8-5	8-4	55
31	Y	0-8	0-8	30	71	↓	11-8-6	0-8-7	56
32	Z	0-9	0-9	31	72	<	12-0 or 12-8-2	12-0 or 12-8-2	72
33	0	0	0	12	73	>	11-8-7	0-8-6	57
34	1	1	1	01	74	≤	8-5	12-8-4	15
35	2	2	2	02	75	≥	12-8-5	0-8-2	75
36	3	3	3	03	76	¬	12-8-6	11-8-7	76
37	4	4	4	04	77	; (semicolon)	12-8-7	11-8-6	77

†This character is lost on even parity magnetic tape.

*Since 00 cannot be represented on magnetic tape, it is converted to BCD 12. On input, it will be translated to display code 33 (number zero).

CDC 63-CHARACTER SET

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	(none)†			16	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	–	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	:(colon)†	8-2	8-2	00*
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	v	11-0 or 11-8-2	11-0 or 11-8-2	52
27	W	0-6	0-6	26	67	∧	0-8-7	12	37
30	X	0-7	0-7	27	70	↑	11-8-5	8-4	55
31	Y	0-8	0-8	30	71	↓	11-8-6	0-8-7	56
32	Z	0-9	0-9	31	72	<	12-0 or 12-8-2	12-0 or 12-8-2	72
33	0	0	0	12	73	>	11-8-7	0-8-6	57
34	1	1	1	01	74	≤	8-5	12-8-4	15
35	2	2	2	02	75	≥	12-8-5	0-8-2	75
36	3	3	3	03	76	¬	12-8-6	11-8-7	76
37	4	4	4	04	77	;(semicolon)	12-8-7	11-8-6	77

†When the 63-Character Set is used, the punch code 8-2 is associated with display code 63, the colon. Display code 00_8 is not included in the 63-Character Set and is not associated with any card punch. The 8-6 card punch (026 keypunch) and the 0-8-4 card punch (029 keypunch) in the 63-Character Set are treated as blank on input.

*Since 00 cannot be represented on magnetic tape, it is converted to BCD 12. On input, it will be translated to display code 33 (number zero).

Diagnostic messages are produced by the FORTRAN Extended compiler during both compilation and execution to inform the user of errors in the source program, input data or intermediate results.

COMPILATION DIAGNOSTICS

Errors detected during compilation are noted on the source listing immediately following the END card. The format of the message is as follows:

```
CARD NO.      SEVERITY           DIAGNOSTIC

   n             e        a      error message
```

n — Card number where error was detected. This number is assigned by the FORTRAN Extended compiler.

e — Indicates the type of diagnostic: I, FC, FE, ANSI

- I — Informative message which indicates minor syntax errors or omissions which have no effect upon compilation or execution.
- FC — When an error of this type is encountered during compilation, the remaining portion of the program is checked for syntax errors only. Program is not executed.
- FE — Error fatal to execution. Program compiles but does not execute.
- ANSI — Usage does not conform to ANSI standards (X3.0 - 1966). ANSI diagnostics are not listed unless the X parameter is specified on the FTN control card.

a — Information in this column will differ according to the type of error encountered. For example, if the same statement label is used more than once, the label number is printed. If a message of the format cn CD n appears, cn is the column number in which the error was detected, and n is the card number.

error message — Error message printed by FORTRAN Extended compiler

Example:

```
    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=SCRT(A)
    103 J=A
    104 DO 1 K=3,J,2
    105 L=I/KEXCEEDS
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

    1        I      START.  ASSUMED PROGRAM NAME WHEN NO HEADER STATEMENT APPEARS
    2       FE      07  CD 3 ZERO LEVEL RIGHT PARENTHESIS MISSING. SCANNING STOPS.
    3       FE              UNRECOGNIZED STATEMENT
    5       FE              DUPLICATE STATEMENT LABEL
    9       FE              SYMBOLIC NAME HAS TOO MANY CHARACTERS
    9       FE       /      THE OPERATOR INDICATED (-,+,*,/, OR **) MUST BE FOLLOWED BY A CONSTANT, NAME, OR LEFT PARENTHESIS.
   11       FE              A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT
   16       FE       7      PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES
   21       FE              UNDEFINED STATEMENT NUMBERS, SEE BELOW

     UNDEFINED LABELS
    3
```

FE	DO LOOPS ARE NESTED MORE THAN 50 DEEP
FC	MISSING OR OUT OF RANGE LABEL ON DO STATEMENT
FE	THE TERMINAL STATEMENT OF THIS DO PRECEDES IT
FE	THE CONTROL VARIABLE OF A DO OR DO IMPLIED LOOP MUST BE A SIMPLE INTEGER VARIABLE
FE	THE SYNTAX OF DO PARAMETERS MUST BE I=M1,M2,M3 OR I=M1,M2
FE	A CONSTANT DO PARAMETER MUST BE BETWEEN 0 AND 131,072
FE	A DO PARAMETER MUST BE A POSITIVE INTEGER CONSTANT OR AN INTEGER VARIABLE
FE	DUPLICATE STATEMENT LABEL
FE	A PREVIOUS STATEMENT MAKES AN ILLEGAL TRANSFER INTO A DO LOOP AT THIS LABEL
FE	THIS STATEMENT MAKES AN ILLEGAL TRANSFER INTO A PREVIOUS DO LOOP
FE	A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT
FE	DO LOOPS TERMINATING ON THIS LABEL ARE IMPROPERLY NESTED
I	THIS STATEMENT REDEFINES A CURRENT DO LOOP CONTROL VARIABLE OR PARAMETER
FE	ENTRY STATEMENTS MAY NOT OCCUR WITHIN THE RANGE OF A DO STATEMENT
FE	DEBUG EXECUTION OPTION SUPPRESSED DUE TO NATURE OF ABOVE FATAL ERRORS
FE	LOOP BEGINNING AT THIS CARD NO IS ENTERED FROM OUTSIDE ITS RANGE AND HAS NO EXITS
FE	UNDEFINED STATEMENT NUMBERS, SEE BELOW
FE	PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USE
I	MORE STORAGE REQUIRED BY DO STATEMENT PROCESSOR FOR OPTIMIZATION
I	THE VARIABLE UPPER LIMIT AND THE CONTROL VARIABLE OF THIS DO ARE THE SAME PRODUCING A NON-TERMINATING LOOP
FC	THIS PROGRAM UNIT HAS TOO MANY DO LOOPS
I	THE CONSTANT LOWER LIMIT IS GREATER THAN THE CONSTANT UPPER LIMIT OF A DO

FE	HEADER CARD SYNTAX ERROR
FE	FILENAME IS GREATER THAN 6 CHARACTERS
FE	FILENAME PREVIOUSLY DEFINED
FE	UNIT NUMBER OR PARITY INDICATOR MUST BE AN INTEGER CONSTANT OR VARIABLE
FE	EQUATED FILENAME NOT PREVIOUSLY DEFINED
FC	TABLES OVERLAP, INCREASE FL
FE	UNRECOGNIZED STATEMENT
FE	ILLEGAL LABEL FIELD IN THIS STATEMENT
FE	STATEMENT TOO LONG
FE	SYMBOLIC NAME HAS MORE THAN 7 CHARACTERS
FE	UNMATCHED PARENTHESIS
ANSI	7 CHARACTER SYMBOLIC NAME IS NON-ANSI
I	NO END CARD, END LINE ASSUMED
FC	TABLE OVERFLOW, INCREASE FL
FE	ILLEGAL CHARACTER. THE REMAINDER OF THIS STATEMENT WILL NOT BE COMPILED.
ANSI	THE FORMAT OF THIS END LINE DOES NOT CONFORM TO ANSI SPECIFICATIONS
FE	RETURNS LIST ERROR
FE	DOUBLY DEFINED DUMMY ARGUMENT
FE	EXECUTABLE STATEMENTS ARE ILLEGAL IN A BLOCK DATA PROGRAM
FE	ILLEGAL SEPARATOR BETWEEN VARIABLES
FE	ARRAY HAS MORE THAN THREE SUBSCRIPTS
FE	ARRAY WITH ILLEGAL SUBSCRIPTS
I	PREVIOUSLY DIMENSIONED ARRAY, FIRST DIMENSIONS WILL BE RETAINED
FE	ARRAY NAME OR VARIABLES USED AS SUBSCRIPTS IN A DIMENSION DECLARATION DO NOT APPEAR AS DUMMY ARGUMENTS
I	CHARACTER BOUNDS REVERSED IN IMPLICIT STATEMENT
FE	ITEMS IN A RETURNS LIST OR EXTERNAL NAMES MAY NOT APPEAR IN DECLARATIVE STATEMENTS
I	PREVIOUSLY TYPED VARIABLE, FIRST TYPE IS RETAINED
FE	DUMMY ARGUMENT IN STATEMENT FUNCTION DEFINITION OCCURRED TWICE

FE	STATEMENT FUNCTION HAS MORE THAN 63 DUMMY ARGUMENTS
FE	SYNTAX ERROR IN STATEMENT FUNCTION DEFINITION
FC	MEMORY OVERFLOW DURING STATEMENT FUNCTION EXPANSION
FC	SYMBOL TABLE OVERFLOW
FE	HEADER CARD NOT FIRST STATEMENT
FE	COMMON BLOCK NAME NOT ENCLOSED IN SLASHES
FE	VARIABLE IN COMMON IS DUMMY ARGUMENT OR PREVIOUSLY DECLARED IN COMMON OR ILLEGAL NAME
FE	ILLEGAL COMMON BLOCK NAME
FE	ILLEGAL SEPARATOR IN EXTERNAL STATEMENT
I	MAY NOT BE USED IN A DEBUG STATEMENT
I	PRESENT USE IN CONTEXT OF THIS NAME DOES NOT MATCH PREVIOUS OCCURRENCES IN DEBUG STMTS
ANSI	IMPLICIT STATEMENT IS NON-ANSI
FE	A REFERENCE TO THIS STATEMENT FUNCTION HAS UNBALANCED PARENTHESIS WITHIN THE PARAMETER LIST
FE	UNMATCHED PARAMETER COUNT IN A REFERENCE TO THIS STATEMENT FUNCTION
FE	ILLEGAL CHARACTER BOUND IN IMPLICIT STATEMENT
FE	A CONSTANT CANNOT BE CONVERTED. CHECK CONSTANT FOR PROPER CONSTRUCT.
I	RETURN STATEMENT APPEARS IN MAIN PROGRAM
FE	RETURNS LIST CANNOT BE USED IN A FUNCTION SUBPROGRAM
FE	ARGUMENT ON NON-STANDARD RETURN STATEMENT IS NOT A RETURNS DUMMY ARGUMENT
FE	A CONSTANT ARITHMETIC OPERATION WILL GIVE AN INDEFINITE OR OUT-OF-RANGE RESULT
FE	ILLEGAL TYPE SPECIFIED IN IMPLICIT STATEMENT
ANSI	THIS RETURN STATEMENT IS NON-ANSI
FE	I/O LIST SYNTAX ERROR
FE	FORMAT REFERENCE MUST BE A LEGAL STATEMENT NUMBER OR AN ARRAY ELEMENT NAME
I	ARRAY REFERENCE OUTSIDE DIMENSION BOUNDS
FE	ECS/LCM REFERENCE MUST BE A STAND-ALONE ARGUMENT TO AN EXTERNAL ROUTINE

I	MISSING I/O LIST OR SPURIOUS COMMA
FE	ENTRY POINT NAMES MUST BE UNIQUE - THIS ONE HAS BEEN PREVIOUSLY USED IN THIS SUBPROGRAM
FE	IMPROPER FORM OF ENTRY STATEMENT, ONLY ALLOWABLE FOR IS [ENTRY NAME]
FE	REFERENCED LABEL IS MORE THAN FIVE CHARACTERS
FE	ENTRY STATEMENT MAY NOT APPEAR IN A MAIN PROGRAM
FE	NAMELIST STATEMENT SYNTAX ERROR
FE	NAMELIST GROUP NAME PREVIOUSLY REFERENCED IN ANOTHER CONTEXT
FE	NAMELIST GROUP NAME NOT ENCLOSED IN SLASHES
FE	APPEARED WHERE A VARIABLE SHOULD HAVE
FE	ILLEGAL NAME USED AS NAMELIST VARIABLE
FE	DUMMY ARGUMENT WITH VARIABLE DIMENSIONS NOT ALLOWED IN A NAMELIST STATEMENT
FE	ENTRY STATEMENT MAY NOT BE LABELED
FE	ILLEGAL SYNTAX IN IMPLICIT STATEMENT
FE	LEVEL 3 VARIABLE MAY NOT APPEAR IN AN EQUIVALENCE STATEMENT
FE	ILLEGAL SUBSCRIPT IN EQUIV STMT
FE	ONLY ONE SYMBOLIC NAME IN EQUIVALENCE GROUP
FE	SYNTAX ERROR IN EQUIVALENCE STATEMENT
FE	DUMMY ARGUMENTS MAY NOT APPEAR IN COMMON OR EQUIV STMTS
FE	COMMON-EQUIVALENCE ERROR
FE	NUMBER OF SUBSCRIPTS IS INCOMPATIBLE WITH THE NUMBER OF DIMENSIONS DURING EQUIVALENCING
FE	ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN
FE	INVOLVED IN CONTRADICTORY EQUIVALENCING
FE	ARRAY OR COMMON VARIABLE MAY NOT BE DECLARED EXTERNAL
ANSI	THIS FORM OF AN I/O STATEMENT DOES NOT CONFORM TO ANSI SPECIFICATIONS
FE	GO TO STATEMENT - SYNTAX ERROR
FE	MISSING STATEMENT LABEL OR SYNTAX ERROR IN COMPUTED OR ASSIGNED GO TO
ANSI	GO TO STATEMENT CONTAINS NON-ANSI USAGES
FE	DEFECTIVE HOLLERITH CONSTANT. CHECK FOR CHARACTER COUNT ERROR, MISSING ≠ DELIMITER OR LOST CONTIN CARD.

I	SINGLE WORD CONSTANT MATCHED WITH DOUBLE OR COMPLEX VARIABLE. PRECISION LOST OR ONLY REAL PART USED
FE	NUMBER OF CHARACTERS IN AN ENCODE/DECODE STATEMENT MUST BE AN INTEGER CONSTANT OR VARIABLE
FE	MORE THAN 50 FILES ON PROGRAM CARD OR 63 PARAMETERS ON A SUBROUTINE OR FUNCTION CARD
FE	DECLARATIVE STATEMENT OUT OF SEQUENCE
FC	ERROR TABLE OVERFLOW
FE	SYNTAX ERROR IN ASSIGN STATEMENT, ONLY ALLOWABLE IS [ASSIGN LABEL TO VARIABLE]
I	VARIABLE LIST EXCEEDS CONSTANT LIST IN DATA STATEMENT, EXCESS VARIABLES NOT INITIALIZED
I	CONSTANT LIST EXCEEDS VARIABLE LIST IN DATA STATEMENT, EXCESS CONSTANTS IGNORED
ANSI	NON-ANSI FORM OF DATA STATEMENT
FE	SYNTAX ERROR IN DATA STATEMENT
FE	SYNTAX ERROR IN DATA CONSTANT LIST
FE	+ OR - SIGN MUST BE FOLLOWED BY A CONSTANT
FE	DATA CONSTANT LISTS MAY ONLY BE NESTED 1 DEEP
FE	CONSTANT IN A DATA STATEMENT MUST BE FOLLOWED BY A , / OR RIGHT PAREN
FE	DO LIMIT OR REP FACTOR MUST BE AN INTEGER OR OCTAL CONSTANT BETWEEN 1 AND 131,072
FE	FOLLOWED BY AN ILLEGAL ITEM
FE	SYNTAX ERROR IN IMPLIED DO NEST
FE	SYNTAX ERROR IN VARIABLE LIST OF DATA STATEMENT
FE	DUPLICATE LOOP INDEX OR DOESNT MATCH ANY SUBSCRIPT VARIABLE
FE	VARIABLE SUBSCRIPTS MAY NOT APPEAR WITHOUT DO LOOPS
FE	VALUE OF ARRAY SUBSCRIPT IS .LT. 1 OR .GT. DIMENSIONALITY IN IMPLIED DO NEST
FE	NON DIMENSIONED NAME APPEARS FOLLOWED BY LEFT PAREN
FE	SYNTAX ERROR IN SUBSCRIPT LIST, MUST BE OF FORM CON1*IVAR+CON2
FE	CONSTANT SUBSCRIPT VALUE EXCEEDS ARRAY DIMENSIONS
FE	ZERO STATEMENT LABELS ARE ILLEGAL
I	CONSTANT LENGTH .GT. VARIABLE LENGTH, CONSTANT TRUNCATED

FE	VARIABLE IN DATA STATEMENT MAY NOT BE FUNCTION NAME, DUMMY ARGUMENT, OR IN BLANK COMMON
FE	THIS NAME MAY NOT BE USED IN A DATA STMT
ANSI	MULTIPLE REPLACEMENT STATEMENT IS NON-ANSI
FE	ILLEGAL USE OF THE EQUAL SIGN
FE	SIMPLE VARIABLE OR CONSTANT FOLLOWED BY LEFT PARENTHESIS
FE	NO MATCHING RIGHT PARENTHESIS
FE	NO MATCHING LEFT PARENTHESIS
FE	-, +, *, /, OR ** MUST BE FOLLOWED BY A CONSTANT, NAME, OR LEFT PARENTHESIS
FE	A NAME MAY NOT BE FOLLOWED BY A CONSTANT
FE	MORE THAN 63 ARGUMENTS IN ARGUMENT LIST
FE	A CONSTANT MAY NOT BE FOLLOWED BY AN EQUAL SIGN, NAME, OR ANOTHER CONSTANT
FE	EXPRESSION TRANSLATOR TABLE (OPSTAK) OVERFLOWED, SIMPLIFY THE EXPRESSION
FE	LOGICAL OPERAND USED WITH NON-LOGICAL OPERATOR
FE	NO MATCHING RIGHT PARENTHESIS IN SUBSCRIPT
I	ARRAY NAME NOT SUBSCRIPTED, FIRST ELEMENT WILL BE USED
FE	INTRINSIC FUNCTION REFERENCE MAY NOT USE A FUNCTION NAME AS AN ARGUMENT
FE	ARGUMENT NOT FOLLOWED BY COMMA OR RIGHT PARENTHESIS
FE	A FUNCTION REFERENCE REQUIRES AN ARGUMENT LIST
FE	SYNTAX ERROR IN CALL STATEMENT
FE	EXPRESSION TRANSLATOR TABLE (FRSTB) OVERFLOWED, SIMPLIFY THE EXPRESSION
FE	A RELATIONAL OPERATOR MUST BE FOLLOWED BY A CONSTANT, NAME, LEFT PAREN., - OR +
I	THE NUMBER OF ARGUMENTS IN THE ARGUMENT LIST OF AN EXTERNAL FUNCTION IS INCONSISTENT
FE	A LIBRARY FUNCTION REFERENCE HAS AN INCORRECT ARGUMENT COUNT
FE	EXPRESSION TRANSLATOR TABLE (ARLIST) OVERFLOWED, SIMPLIFY THE EXPRESSION
ANSI	ARRAY NAME REFERENCED WITH FEWER SUBSCRIPTS THAN DIMENSIONALITY OF ARRAY IS NON-ANSI

FE	ILLEGAL LIST ITEM ENCOUNTERED IN AN I/O LIST
FE	RIGHT PARENTHESIS FOLLOWED BY A NAME, CONSTANT OR LEFT PARENTHESIS
FE	MORE THAN ONE RELATIONAL OPERATOR IN A RELATIONAL EXPRESSION
FE	A COMMA, LEFT PAREN., =, .OR., OR .AND. MUST BE FOLLOWED BY A NAME, CONSTANT, LEFT PAREN., -, .NOT., OR +
FE	AN ARRAY REFERENCE HAS TOO MANY SUBSCRIPTS
FE	MISSING RIGHT PARENTHESIS IN ARGUMENT LIST
FE	ILLEGAL FORM INVOLVING THE USE OF A COMMA
FE	LOGICAL AND NON-LOGICAL OPERANDS MAY NOT BE MIXED
FE	DIVISION BY ZERO
FE	A COMPLEX BASE MAY ONLY BE RAISED TO AN INTEGER POWER
FE	ILLEGAL USE OF THIS PROGRAM OR SUBROUTINE NAME IN AN EXPRESSION
FE	SUBROUTINE NAME REFERRED TO BY CALL IS USED ELSEWHERE AS A NON-SUBROUTINE NAME
I	THE NUMBER OF ARGUMENTS IN A SUBROUTINE ARGUMENT LIST IS INCONSISTENT
FE	ONE OF THE ARGUMENTS IN A RETURNS LIST IS ILLEGAL
FE	ILLEGAL LABELS IN IF STATEMENT
FE	LOGICAL EXPRESSION IN 3-BRANCH IF STATEMENT
FE	THE STATEMENT IN A LOGICAL IF MAY BE ANY EXECUTABLE STATEMENT OTHER THAN A DO OR ANOTHER LOGICAL IF
FE	THE EXPRESSION IN A LOGICAL IF IS NOT TYPE LOGICAL
I	THERE IS NO PATH TO THIS STATEMENT
FE	VARIABLE IN ASSIGN OR ASSIGNED GO TO MUST BE INTEGER VARIABLE
ANSI	NAMELIST STATEMENT IS NON-ANSI
ANSI	ENTRY STATEMENT IS NON-ANSI
ANSI	RETURNS PARAMETERS IN CALL STATEMENT IS NON-ANSI
ANSI	NON-ANSI USE OF HOLLERITH CONSTANT
I	A HOLLERITH CONSTANT IS AN OPERAND OF AN ARITHMETIC OPERATOR
ANSI	THIS SUBSCRIPT IS NON-ANSI
ANSI	MASKING EXPRESSION IS NON-ANSI
ANSI	THIS COMBINATION OF TYPES IN EXPONENTIATION IS NON-ANSI

ANSI	A COMPLEX OPERAND IN A RELATIONAL EXPRESSION IS NON-ANSI
ANSI	THIS COMBINATION OF TYPES USED WITH A RELATIONAL OR ARITHMETIC OPERATOR (OTHER THAN **) IS NON-ANSI
ANSI	THIS COMBINATION OF TYPES IN AN ASSIGNMENT STATEMENT IS NON-ANSI
ANSI	TWO-BRANCH IF STATEMENT IS NON-ANSI
ANSI	A TYPE COMPLEX EXPRESSION IN AN IF STATEMENT IS NON-ANSI
I	THIS STATEMENT BRANCHES TO ITSELF
I	THIS IF DEGENERATES INTO A SIMPLE TRANSFER TO THE LABEL INDICATED
FE	TOO MANY SUBSCRIPTS IN ARRAY REFERENCE
ANSI	LOGICAL OPERATOR OR CONSTANT USAGE IS NON-ANSI
ANSI	OCTAL CONSTANT OR R,L FORMS OF HOLLERITH CONTANT IS NON-ANSI
FE	LEFT SIDE OF ASSIGNMENT STATEMENT IS ILLEGAL
FE	REFERENCE TO STATEMENT FUNCTION HAS AN ARGUMENT MISSING
FE	ALL ELEMENTS OF THIS COMMON BLOCK MUST BE LEVEL 3
FE	A PREVIOUSLY MENTIONED ADJUSTABLE SUBSCRIPT IS NOT TYPE INTEGER
FE	ALL LEVEL 3 ITEMS MUST APPEAR IN A LABELED COMMON BLOCK
FE	THE TYPE OF THIS IDENTIFIER IS NOT LEGAL FOR ANY EXPRESSION
FE	A CONSTANT IN A REAL EXPRESSION IS OUT OF RANGE OR INDEFINITE
I	ONLY THOSE ERRORS WHICH ARE FATAL TO EXECUTION WILL BE LISTED BEYOND THIS POINT
FE	WAS LAST CHARACTER SEEN AFTER TROUBLE, REMAINDER OF STATEMENT IGNORED
ANSI	DOLLAR SIGN STATEMENT SEPARATOR IS NON-ANSI
ANSI	AN EXPRESSION IN AN OUTPUT STATEMENT I/O LIST IS NON-ANSI
FE	FIRST WORD AND LAST WORD ADDRESSES OF DATA TRANSMISSION BLOCK MUST BE IN THE SAME LEVEL
FE	THE VALUE OF THE PARITY INDICATOR IN A BUFFER I/O STATEMENT MUST BE 0 OR 1
ANSI	ARRAY NAME OPERAND NOT SUBSCIPTED, FIRST ELEMENT WILL BE USED
I	MASK ARGUMENT OUT OF RANGE. A MASK OF 0 OR 60 WILL BE SUBSTITUTED FOR ARGUMENT.
FE	SYNTAX ERROR IN STATEMENT FUNCTION DEFINITION
I	ASSUMED PROGRAM NAME WHEN PROGRAM STATEMENT MISSING

I	NUMBER OF DIGITS IN CONSTANT EXCEED POSSIBLE SIGNIFICANCE. HIGH ORDER DIGITS RETAINED IF POSSIBLE.
FE	.NOT. MAY NOT BE PRECEDED BY NAME, CONSTANT, OR RIGHT PARENS
FE	THE FIELD FOLLOWING STOP OR PAUSE MUST BE 5 OR LESS OCTAL DIGITS OR A QUOTE-DELIMITED STRING
FE	ILLEGAL VARIABLE IN ASSIGN OR ASSIGNED GOTO
ANSI	THIS FORMAT DECLARATION IS NON-ANSI
FE	NO TERMINATING RIGHT PARENTHESIS IN OVERLAY LOADER DIRECTIVE
FC	TOO MUCH OVERLAY CONTROL CARD INFORMATION - INCREASE FL
FE	ONLY ONE ECS COMMON BLOCK MAY BE DECLARED
ANSI	END STATEMENT ACTING AS A RETURN IS NON-ANSI
I	ARRAY IS ENLARGED BY EQUIVALENCING
FE	TOO MANY LABELED COMMON BLOCKS, ONLY 125 BLOCK ARE ALLOWED
FE	UNIT NUMBER MUST BE BETWEEN 1 AND 99 INCLUSIVE
FE	FUNCTION NAME DOES NOT APPEAR AS A VARIABLE IN THIS SUBPROGRAM
FE	SUBROUTINE NAME MUST NOT APPEAR IN A SPECIFICATION STATEMENT
I	BUFFER LENGTH REQUESTED IS TOO LARGE. STANDARD LENGTH OF 2000B SUBSTITUTED.
I	/ OR COMMA MISSING. COMMA ASSUMED HERE.
I	X-FIELD PRECEDED BY A BLANK. 1X ASSUMED.
I	X-FIELD PRECEDED BY A ZERO, NO SPACING OCCURS
I	PRECEDING FIELD WIDTH IS ZERO
I	PRECEDING FIELD WIDTH SHOULD BE 7 OR MORE
I	FLOATING POINT DESCRIPTOR EXPECTS DECIMAL POINT SPECIFIED. OUTPUT WILL INCLUDE NO FRACTIONAL PARTS.
I	FLOATING POINT SPECIFICATION EXPECTS DECIMAL DIGITS TO BE SPECIFIED. ZERO DECIMAL DIGITS ASSUMED.
I	REPEAT COUNT FOR PRECEDING FIELD DESCRIPTOR IS ZERO
I	IF FORMAT IS USED FOR OUTPUT AN ERROR WILL OCCUR
I	PRECEDING SCALE FACTOR IS OUTSIDE LIMITS OF REPRESENTATION WITHIN THE MACHINE
I	SUPERFLUOUS P SPECIFICATION

I	RECORD SIZE TOO SMALL. CHECK USE OF THIS FORMAT TO ENSURE DEVICE CAN HANDLE THIS RECORD SIZE.
I	EW.D OR DW.D DESCRIPTOR BAD FOR OUTPUT, W SHOULD SATISFY W-7 .GE. D
I	NUMERIC FIELD FOLLOWING T SPECIFICATION IS EQUAL TO ZERO, COLUMN ONE IS ASSUMED
I	P SPECIFICATION HAS NUMBER MISSING - 0P IS ASSUMED
I	NON-BLANK CHARACTERS FOLLOW OUTER RIGHT PARENTHESIS. THESE CHARACTERS WILL BE IGNORED.
I	TAB SETTING MAY EXCEED RECORD SIZE, DEPENDING ON USE
ANSI	PLUS SIGN IS AN ILLEGAL CHARACTER
ANSI	PRECEDING FIELD DESCRIPTOR IS NON-ANSI
ANSI	FLOATING PT DESCRIPTOR EXPECTED FOLLOWING P SPECIFICATION
ANSI	T SPECIFICATION IS NON-ANSI
ANSI	HOLLERITH STRING DELINEATED BY SYMBOLS IS NON-ANSI
FE	PRECEDING CHARACTER ILLEGAL AT THIS POINT IN CHARACTER STRING. SCAN OF THIS FORMAT STOPS HERE.
FE	ILLEGAL CHARACTER FOLLOWS PRECEDING FLOATING PT DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.
FE	ILLEGAL CHARACTER FOLLOWS PRECEDING A, I, L, O, OR R DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.
FE	ILLEGAL CHARACTER FOLLOWS T SPECIFICATION. ERROR SCAN FOR THIS FORMAT STOPS HERE.
FE	ILLEGAL CHARACTER FOLLOWS SIGN CHARACTER. ERROR SCANNING FOR THIS FORMAT STOPS HERE.
FE	PRECEDING CHARACTER ILLEGAL. SCALE FACTOR EXPECTED. ERROR SCANNING FOR THIS FORMAT STOPS HERE.
FE	PRECEDING HOLLERITH COUNT IS EQUAL TO ZERO. ERROR SCANNING FOR THIS FORMAT STOPS HERE.
FE	FORMAT STATEMENT ENDS BEFORE LAST HOLLERITH COUNT IS COMPLETE. ERROR SCAN FOR THIS FORMAT STOPS AT H.
FE	FORMAT STATEMENT ENDS BEFORE END OF HOLLERITH STRING. ERROR SCANNING STOPS HERE.
FE	PRECEDING HOLLERITH INDICATOR IS NOT PRECEDED BY A COUNT. SCANNING STOPS HERE.
FE	OUTER RIGHT PARENTHESIS MISSING. SCANNING STOPS.
FE	MAXIMUM PARENTHESIS NESTING LEVEL EXCEEDED. ERROR SCAN OF THIS FORMAT STOPS HERE.

FE	FIELD WIDTH TOO LARGE FOR RECORD SIZE. SCANNING CONTINUES.
FE	RECORD LENGTH OUTSIDE LIMITS FOR RECORD SIZE. SCANNING CONTINUES.
FE	T SPECIFICATION IS TOO LARGE FOR RECORD LENGTH. SCANNING CONTINUES.
I	COMMA MISSING BEFORE VARIABLE INDICATED
FE	ILLEGAL SEPARATOR ENCOUNTERED
FE	SYNTAX ERROR
FE	LEVEL 3 COMMON BLOCKS MUST BE LABELED
FE	CONSTANT TABLE CONSTORS OVERFLOWED - STATEMENT TRUNCATED. ENLARGE TABLE OR SIMPLIFY STATEMENT.
FE	INVALID LEVEL NUMBER SPECIFIED
I	LEVEL CONFLICTS WITH PREVIOUS DECLARATION. ORIGINAL LEVEL RETAINED.
FE	CONFLICTING LEVEL DECLARATIONS EXIST IN THIS COMMON BLOCK
I	NOT ALL ITEMS IN THIS COMMON BLOCK OCCUR IN LEVEL STATEMENTS
FE	ITEMS IN DIFFERENT LEVELS OF STORAGE MAY NOT BE EQUIVALENCED
I	THE ECS STATEMENT IS OBSOLETE. USE A LEVEL 3 STATEMENT.
FE	ARG TO LOCF MAY NOT BE AN EXPRESSION

EXECUTION DIAGNOSTICS

Execution errors are fatal unless a non-standard recovery routine is specified by the user (refer to section 3, part 3).

Execution diagnostics are printed on the source listing in the following format:

```
ERROR NUMBER x DETECTED BY routine AT ADDRESS y

CALLED FROM routine AT ADDRESS z

or CALLED FROM routine AT LINE d

ERROR SUMMARY

ERROR    TIMES

 x         n
```

y and z are addresses, x is a decimal error number, and d is a line number as printed on the source listing. n is a decimal digit which indicates the number of times the error occurred.

Example:

```
      PROGRAM EXERR(INPUT,OUTPUT)
      N=5
      GO TO (1,2,3),N
    1 N=N+1
    2 N=N+2
    3 STOP
      END
```

```
ERROR IN COMPUTED GOTO STATEMENT- INDEX VALUE INVALID
ERROR NUMBER   0001 DETECTED BY ACGOER  AT ADDRESS 000001
CALLED FROM EXERR   AT  LINE 0003
```

In the following list of execution diagnostics:

R = an argument

I and J = integer

Y and X = real

D = double precision

Z = complex

Error No.	Message	Routine
1	ERROR IN COMPUTED GO TO STATEMENT - INDEX VALUE INVALID	ACGOER$
2	ABS(R).GT.1.0 INFINITE ARGUMENT INDEFINITE ARGUMENT	ACOS
3	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	ALOG
4	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	ALOG10
5	ABS(R).GT.1.0 INFINITE ARGUMENT INDEFINITE ARGUMENT	ASIN
6	INFINITE ARGUMENT INDEFINITE ARGUMENT	ATAN
7	X=Y=0.0 INFINITE ARGUMENT INDEFINITE ARGUMENT	ATAN2
8	FLOATING OVERFLOW INFINITE ARGUMENT INDEFINITE ARGUMENT	CABS
9	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ZTOI
10	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CCOS
11	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CEXP

12	ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	CLOG
13	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	COS
14	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CSIN
15	INFINITE ARGUMENT INDEFINITE ARGUMENT	CSQRT
16	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOX (D**X)
17	INFINITE ARGUMENT INDEFINITE ARGUMENT	DATAN
18	X=Y=0.0 INFINITE ARGUMENT INDEFINITE ARGUMENT	DATAN2
19	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOD (D**D)
20	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOI (D**I)
21	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 ARGUMENT INDEFINITE ARGUMENT	DTOZ (D**Z)
22	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	DCOS
23	ARGUMENT TOO LARGE, FLOATING OVERFLOW INFINITE ARGUMENT INDEFINITE ARGUMENT	DEXP

24	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DLOG
25	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DLOG10
26	DOUBLE PRECISION INTEGER EXCEEDS 96 BITS 2ND ARGUMENT ZERO INFINITE ARGUMENT INDEFINITE ARGUMENT	DMOD
28	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	DSIN
29	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DSQRT
30	ARGUMENT TOO LARGE, FLOATING OVERFLOW INFINITE ARGUMENT INDEFINITE ARGUMENT ARGUMENT TOO SMALL	EXP
31	INTEGER OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER	ITOJ
33	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOD (X**D)
34	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOI (X**I)
35	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOY (X**Y)
36	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	SIN

37	ILLEGAL SENSE LITE NUMBER	SLITE
38	ILLEGAL SENSE LITE NUMBER	SLITET
39	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	SQRT
40	ILLEGAL SENSE SWITCH NUMBER	SSWTCH
41	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	TAN
42	INFINITE ARGUMENT INDEFINITE ARGUMENT	TANH
43	MASK OUT OF RANGE	MASK
44	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ITOD (I**D)
45	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ITOX (I**X)
46	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 ARGUMENT INDEFINITE ARGUMENT	ITOZ (I**Z)
47	FLOATING OVERFLOW IN X**REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(X) TOO LARGE INFINITE ARGUMENT	XTOZ
48	FATAL ERROR ENCOUNTERED DURING PROGRAM EXECUTION DUE TO COMPILATION ERROR	FTNERR$
49	TOO FEW CONSTANTS FOR UNSUBSCRIPTED ARRAY	NAMEIN=
50	FATAL ERROR IN LOADER	OVERLA$
55	END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx	BUFIN=

56	WRITE FOLLOWED BY READ ON FILE-xxxxxxx	
57	BUFFER DESIGNATION BAD--FWA.GT.LWA	
59	BUFFER SPECIFICATION BAD--FWA.GT.LWA	BUFOUT=
62	FILENAME NOT DECLARED-xxxxxxx	GETFIT$
63	END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx	INPUTB=
65	END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx	INPUTC=
66	PRECISION LOST IN FLOATING INTEGER CONSTANT	NAMEIN=
	NAMELIST DATA TERMINATED BY EOF, NOT $	
	NAMELIST NAME NOT FOUND	
	NO I/O MEDIUM ASSIGNED	
	WRONG TYPE CONSTANT	
	INCORRECT SUBSCRIPT	
	TOO MANY CONSTANTS	
	(,$, OR = FXPECTED, MISSING	
	VARIABLE NAME NOT FOUND	
	BAD NUMERIC CONSTANT	
	MISSING CONSTANT AFTER *	
	UNCLEARED EOF ON A READ	
	READ PARITY ERROR	
67	DECODE CHARACTER RECORD .GT. 150	DECODE=
68	*ILLEGAL FUNCTIONAL LETTER	KODER$
69	*IMPROPER PARENTHESIS NESTING	
70	*EXCEEDED RECORD SIZE	
71	*SPECIFIED FIELD WIDTH ZERO	
72	*FIELD WIDTH .LE. DECIMAL WIDTH	
73	*HOLLERITH FORMAT WITH LIST	
74	*ILLEGAL FUNCTIONAL LETTER	KRAKER$
75	*IMPROPER PARENTHESIS NESTING	
76	*SPECIFIED FIELD WIDTH ZERO	
77	*EXCEEDED RECORD SIZE	
78	*ILLEGAL DATA IN FIELD * *	
79	*DATA OVERFLOW *>*	
80	*HOLLERITH FORMAT WITH LIST	
83	OUTPUT FILE LINE LIMIT EXCEEDED	OUTPTC=
84	OUTPUT FILE LINE LIMIT EXCEEDED	NAMOUT=
85	ENCODE*CHAR/REC .GT. 150*	ENCODE=
87	*LIST/FMT CONFLICT, SINGLE/DOUBLE	KODER
88	WRITE FOLLOWED BY READ ON FILE-xxxxxxx	INPUTB=
89	LIST EXCEEDS DATA, FILENAME-xxxxxxx	
90	PARITY ERROR READING (BINARY) FILE-xxxxxxx	
91	WRITE FOLLOWED BY READ ON FILE-xxxxxxx	INPUTC=
92	PARITY ERROR READING (CODED) FILE-xxxxxxx	
93	PARITY ERROR ON LAST READ ON FILE-xxxxxxx	OUTPTB=

94	PARITY ERROR ON LAST READ ON FILE-xxxxxxx	OUTPTC=
97	INDEX NUMBER ERROR	RANMS=
98	FILE ORGANIZATION OR RECORD TYPE ERR	
99	WRONG INDEX TYPE	
100	INDEX IS FULL	
101	DEFECTIVE INDEX CONTROL WORD	
102	RECORD LENGTH EXCEEDS SPACE ALLOCATED	
103	6RM/7DM I/O ERR NUMBER 000	
104	INDEX KEY UNKNOWN	
112	ECS UNIT HAS LOST POWER OR IS IN MAINTENANCE MODE	WRITEC
113	ECS READ PARITY ERROR	READEC

SYSTEM ROUTINE SPECIFICATIONS III-3

The SYSTEM routine handles error tracing, diagnostic printing, termination of output buffers, and transfer to specified non-standard error procedures. All FORTRAN mathematical routines rely on SYSTEM to complete these tasks; also, a FORTRAN coded routine may call SYSTEM. Any argument 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 following routines must always be available, they are combined into one subprogram with multiple entry points.

Q8NTRY.	Initializes input/output buffer parameters
STOP.	Enters STOP in dayfile and begins END processing
EXIT.	Enters EXIT in dayfile and begins END processing
END.	Terminates all output buffers, prints an error summary, transfers control to the main overlay if within an overlay; in any other case exits to monitor.
SYSTEM	Handles error tracing, diagnostic printing, termination of output buffers; and depending on type of error, transfers to specified non-standard error recovery address, terminates the job, or returns to calling routine.
SYSTEMC	Changes entry to SYSTEM's error table according to arguments passed.

CALLING SYSTEM

The calling sequence to SYSTEM passes the error number as the first argument and an error message as the second argument; therefore, several messages may be associated with one error number. The error summary at program termination lists the total number of times each error number was encountered.

ERROR PROCESSING

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 line printed 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 that exceeds the table length. An entry in the error table appears as follows:

Print Frequency	Frequency Increment	Print Limit	Detection Total	F/NF	A/NA	Non-standard Recovery Address
8	8	12	12	1	1	18

Print frequency is used as follows:

Print Frequency	Increment	
0	0	Diagnostic and traceback information are not listed.
0	1	Diagnostic and traceback information are listed until the print limit is reached.
0	n	Diagnostic and traceback information are listed only the first n times unless the print limit is reached first.
n		Diagnostic and traceback information are listed every nth time until the print limit is reached.

STANDARD RECOVERY

If the error is non-fatal (NF), and no non-standard recovery address is specified; error messages are printed according to print frequency, and control is returned to the calling routine.

If the error is fatal (F), 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.

NON-STANDARD RECOVERY

If a non-standard recovery is specified, SYSTEM supplies the recovery routine with the following information:

A1	Address of argument list passed to routine detecting the error
X1	Address of first argument in the list
A0	Address of argument list of routine that called the routine detecting the error
B1	Address of a secondary argument list which contains, in successive words: Error number passed to 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 traceback information in lower 30 bits for routine that called SYSTEM
A2	Address of error table entry in SYSTEM
X2	Contents of error table entry

Information in the secondary argument list is not available to FORTRAN-coded routines.

NON-FATAL ERROR

The routine detecting the error and SYSTEM are delinked from the calling chain, and the non-standard recovery routine is entered. When the recovery routine exits in the normal manner, control returns to the routine that called the routine detecting the error.

Thus, any faulty arguments can 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 and returns control to SYSTEM, the recovery routine is free to continue computation.

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 argument 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 call is made to SYSTEMC with the following arguments:

Error number

List containing 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 3 = 0 never list error word 3 = 1 always list error 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 in the argument list is negative, the value already in table entry is not to be altered.

(Since the auxiliary table bit can be set only during assembly of SYSTEM, only then can an auxiliary table entry be made.)

FILE NAME HANDLING BY SYSTEM

The file names in the PROGRAM statement are placed in RA+2 and the locations immediately following by SYSTEM (Q8NTRY). RA is the reference address, the absolute address where the user's field length begins. The file name is left justified, and the file's file information table (FIT) address is right justified in the word.

The logical file name (LFN) which appears in the first word of the file information table is determined in one of three ways:

1. If no arguments are specified on the load card, the logical file name is the file name in the PROGRAM statement.

   Example:

```
FTN.
LGO.
 .
 .
 .
PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)
```

   Contents of RA+2 before execution of Q8NTRY:

   000 ... 000
   000 ... 000

Contents of RA+2 after execution of Q8NTRY:	The logical file names in the file information table will be:
INPUT ... fit address	INPUT
OUTPUT .. fit address	OUTPUT
TAPE1 ... fit address	TAPE1
TAPE2 ... fit address	TAPE2

2. If file names are specified on the load card, the logical file name is the name specified on the load card. If no file names are specified on the load card, it is the file name from the PROGRAM statement. A one-to-one correspondence exists between parameters on the load card and parameters in the PROGRAM statement.

   Example:

```
FTN.
LGO(,,DATA,ANSW)
 .
 .
 .
PROGRAM TEST2(INPUT,OUTPUT,TAPE1,TAPE2,TAPE3=TAPE1)
```

Contents of RA+2 before execution of Q8NTRY:

000 ... 000
000 ... 000
DATA .. 000
ANSW .. 000

Contents of RA+2 after execution of Q8NTRY:	The logical file names in the file information table will be:
INPUT ... fit address	INPUT
OUTPUT .. fit address	OUTPUT
TAPE1 ... fit address	DATA
TAPE2 ... fit address	ANSW
TAPE3 ... fit address of TAPE1	uses TAPE1 file information table

3. If a file name in the PROGRAM statement is equivalenced, the logical file name is the file to the right of the equal sign. No new file information table is created.

   Example:

```
FTN.
LGO(,,DATA,ANSW)
 .
 .
 .
PROGRAM TEST3(INPUT,OUTPUT,TAPE1=OUTPUT,TAPE2,TAPE3)
```

   Contents of RA+2 before execution of Q8NTRY:

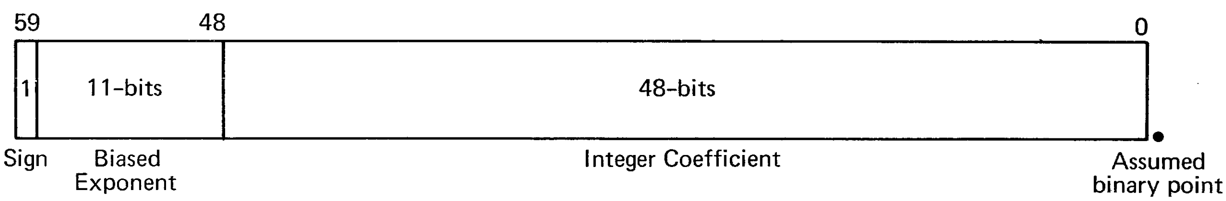
   000 ... 000
   000 ... 000
   DATA .. 000
   ANSW .. 000

Contents of RA+2 after execution of Q8NTRY:	The logical file names in the file information table will be:
INPUT ... fit address	INPUT
OUTPUT .. fit address	OUTPUT
TAPE1 ... fit address of OUTPUT	uses OUTPUT file information table
TAPE2 ... fit address	ANSW
TAPE3 ... fit address	TAPE3

ARITHMETIC III-4

FLOATING POINT ARITHMETIC

Floating point arithmetic is carried out in the functional units of the central processor.



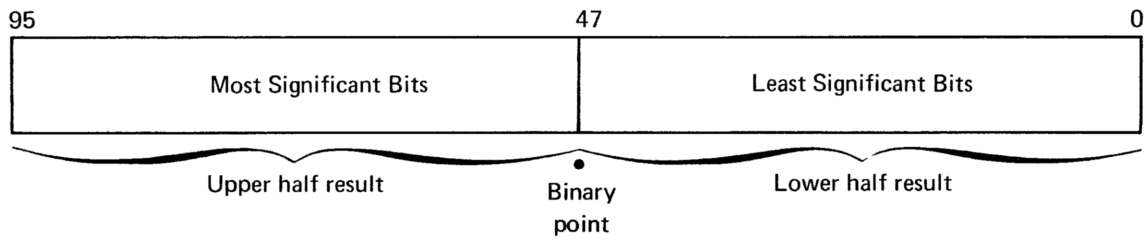
In the 60-bit floating point format shown above, the binary point is considered to be to the right of the coefficient. The lower 48 bits express the integer coefficient, which is the equivalent of approximately 14 decimal digits. The sign of the number is the highest order bit of the packed word. Negative numbers are represented by the one's complement of the 60-bit number.

The exponent portion of the floating point format is biased by 2000 octal. This particular format for floating point numbers was chosen so that the packed form may be treated as a 60-bit integer for sign, equality and zero tests. (Refer to 6400/6500/6600 Computer Systems Reference Manual or 7600 Computer System Reference Manual for details of the hardware pack instruction.)

The following table summarizes the configurations of bits 58 and 59 and the signs of the possible combinations. The number is negative if bit 59 is 1 and positive if bit 59 is 0.

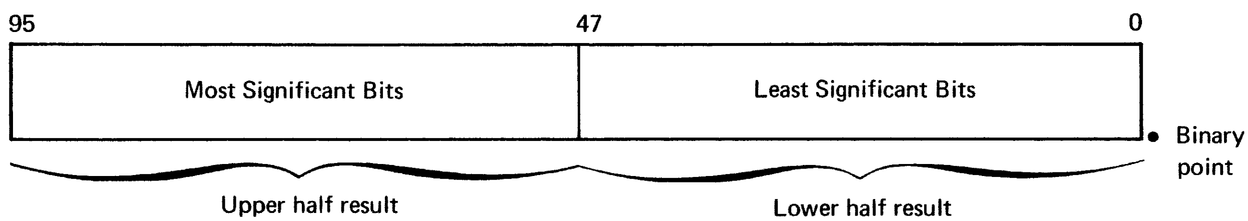
Bit 59	Coefficient Sign	Bit 58	Exponent Sign
0	Positive	1	Positive
0	Positive	0	Negative
1	Negative	0	Positive
1	Negative	1	Negative

To add or subtract two floating point numbers, the floating point ADD unit enters the coefficient with the smaller exponent into the upper half of an accumulator and shifts it right by the difference of the exponents. Then it adds the other coefficient into the upper half of the accumulator. The result is a double length register with the following format:



If single precision is selected, the result is the upper 48 bits of the 96-bit result and the larger exponent. Selecting double precision causes the lower 48 bits of the 96-bit result and the larger exponent minus 60 octal (or 48) to be returned as the result. The subtraction of 60 octal (or 48) is necessary because effectively, the binary point is moved from the right of bit 48 to the right of bit 0.

The multiply units generate 96-bit products from two 48-bit coefficients. The result of a multiply operation is a double length register with the following format:



When unrounded instructions are used, the upper and lower half results with proper exponents may be recovered separately; when rounded instructions are used, only upper half results may be obtained.

If single precision is selected, the upper 48 bits of the product and the sum of the exponents plus 60 octal (or 48) are returned as the result. The addition of 60 octal (or 48) is necessary because, effectively, the binary point is moved from the right of bit 0 to the right of bit 48 when the upper half of the 96-bit result is selected. If double precision is selected, the lower 48 bits of the product and the sum of the exponents is the result.

Some examples of floating point numbers are shown below in octal notation.

Normalized floating point +1	= 1720 4000 0000 0000 0000
Normalized floating point +100	= 1726 6200 0000 0000 0000
Normalized floating point -100	= 6051 1577 7777 7777 77777
Normalized floating point 10^{+64}	= 2245 6047 4037 2237 7733
Normalized floating point 10^{-65}	= 6404 2570 0025 6605 5317

OVERFLOW ($+\infty$ or $-\infty$)

Overflow of the floating point range is indicated by an exponent of 3777 for a positive result and 4000 for a negative result. These are the largest exponent values that can be represented in floating point format, as shown in the floating point table. If the computed value of an exponent is exactly 3777 or 4000, a partial overflow condition exists. The error mode 2 flag is not set by a partial overflow. However, any further computation in floating point functional units with this exponent will set an error mode 2 flag. A complete overflow occurs when a floating point functional unit computes a result that requires an exponent larger than 3777 or 4000.

In this case the result is given a 3777 or 4000 exponent and a zero coefficient. The sign of the coefficient remains the same, as if the result had not exceeded the floating point range. Any further computation in floating point functional units with this result sets an error mode 2 flag.

In this case, the result is given a 3777 or 4000 exponent and a zero co-efficient. The sign of the coefficient remains the same, as if the result had not exceeded the floating point range. The coefficient calculation is ignored, and the overflow condition flag is set in the Program Status Designator (PSD) register. When the overflow condition occurs, the overflow flag in the PSD register causes an error mode 2 message to be printed. Printing an error mode 2 message is the default condition; alternative actions can be specified by the user (refer to SCOPE Reference Manual).

UNDERFLOW (+0 or -0)

Underflow of the floating point range is indicated by an exponent of 0000 for positive numbers and 7777 for negative numbers, the smallest exponent values that can be represented in floating point format. If these exponent values happen to be the exact representation of a result, a partial underflow condition exists; and the underflow condition flag is not set. However, further computation in floating point functional units with these exponents may set the underflow condition flag.

A complete underflow occurs when a floating point functional unit computes a result that requires an exponent smaller than 0000 or 7777. In this case the result is given a 0000 or 7777 exponent and zero coefficient. The sign of the coefficient will be the same as that generated if the result had not fallen short of the floating point range. Thus, the complete underflow indicator is a word of all zero bits, or all one bits, depending on the sign. It is the same as a zero word in integer format.

No underflow indicator is set and no error message is printed.

A complete underflow occurs for this instruction whenever the exponent computation results in less than -1776 octal. This situation is sensed as a special case, and a complete zero word with proper sign results; the coefficient calculation is ignored, and the underflow condition flag is set in the PSD register. When the underflow condition occurs, the underflow flag in the PSD register causes an error mode 2 message to be printed. Printing an error mode 2 message is the default condition; alternative actions can be specified by the user (refer to SCOPE Reference Manual).

INDEFINITE RESULT

An indefinite result indicator is generated by a floating point functional unit when a calculation cannot be resolved; such as a division operation where the divisor and the dividend are both zero. Another case is multiplication of an overflow number times zero. An indefinite result is a value which cannot occur in normal floating point calculations. An indefinite result is represented by a minus zero exponent and a zero coefficient (17770 --- 0).

Any floating point functional unit receiving an indefinite indicator as an operand will generate an indefinite result regardless of the other operand value, and set an error mode 4 flag.

FLOATING POINT REPRESENTATION TABLE

	Positive Coefficient		Negative Coefficient	
OVERFLOW	Complete Overflow Partial Overflow	= 3777 0- - - - 0 = 3777 X- - - - X	Complete Overflow Partial Overflow	= 4777 7- - - - 7 = 4000 X- - - - X
INTEGERS	Largest: 7- - - - 7. x 2^{+1776} Smallest: 1. x 2^0	= 3776 7- - - - 7 = 2000 0- - - 01	*Largest: -7- - - - 7. x 2^{-1776} *Smallest: -1. x 2^0	= 4001 0- - - - 0 = 5777 7- - - 76
ZERO	Positive Zero	= 2000 0- - - - 0	Negative Zero	= 5777 7- - - - 7
INDEFINITE OPERANDS	Indefinite Operand	= 1777 0- - - - 0	**Indefinite Operand	= 6000 7- - - - 7
FRACTIONS	Largest: 7- - - - 7. x 2^{-60} Smallest: 1. x 2^{-1777}	= 1717 7- - - - 7 = 0000 0- - - 01	*Largest: -7- - - - 7. x 2^{-60} *Smallest: -1. x 2^{-1777}	= 6060 0- - - - 0 = 7777 7- - - 76
UNDERFLOW	Complete Underflow Partial Underflow	= 0000 0- - - - 0 = 0000 X- - - - X	Complete Underflow Partial Underflow	= 7777 7- - - - 7 = 7777 X- - - - X

* In absolute value.

** An indefinite operand with a negative sign can occur only from packing or Boolean operations.

NON-STANDARD FLOATING POINT ARITHMETIC

Indefinite result indications:

0000X------X = positive zero (+0)
7777X------X = negative zero (-0)
3777X------X = positive infinity ($+\infty$)
4000X------X = negative infinity ($-\infty$)
1777X------X = positive indefinite (+IND)
6000X------X = negative indefinite (-IND)

where X is an unspecified octal digit.

If the correct result of an operation coincides with any of the above exponents, no error flag is set.

When a floating point arithmetic unit uses one of these six special forms as an operand, however, only the following octal words can occur as results and the associated error mode flag is set.

37770------0 = positive infinity ($+\infty$)	Overflow condition flag
40007------7 = negative infinity ($-\infty$)	Overflow condition flag
17770------0 = positive indefinite (+IND)	Indefinite condition flag
00000------0 = positive zero (+0)	Underflow condition flag

The following tabulations show results of the add, subtract, multiply and divide operations using various combinations of infinite, indefinite, and zero quantities as operands. The designations w and n are defined as follows:

w = any word except $\pm\infty$, IND
n = any word except $\pm\infty$, IND, or ± 0

ADD
X1 = X2 + X3

X3

X2	W	$+\infty$	$-\infty$	$\pm$IND
W	-	$+\infty$	$-\infty$	IND
$+\infty$	$+\infty$	$+\infty$	IND	IND
$-\infty$	$-\infty$	IND	$-\infty$	IND
$\pm$IND	IND	IND	IND	IND

SUBTRACT
X1 = X2 - X3

X3

X2		W	+∞	-∞	±IND
	W	-	-∞	+∞	IND
	+∞	+∞	IND	+∞	IND
	-∞	-∞	-∞	IND	IND
	±IND	IND	IND	IND	IND

MULTIPLY
X1 = X2 * X3

X3

X2		+N	-N	+0	-0	+∞	-∞	±IND
	+N	-	-	0	0	+∞	-∞	IND
	-N	-	-	0	0	-∞	+∞	IND
	+0	0	0	0	0	IND	IND	IND
	-0	0	0	0	0	IND	IND	IND
	+∞	+∞	-∞	IND	IND	+∞	-∞	IND
	-∞	-∞	+∞	IND	IND	-∞	+∞	IND
	±IND	IND	IND	IND	IND	IND	IND	IND

DIVIDE
X1 = X2 / X3

X3

X2		+N	-N	+0	-0	+∞	-∞	±IND
	+N	-	-	+∞	-∞	0	0	IND
	-N	-	-	-∞	+∞	0	0	IND
	+0	0	0	IND	IND	0	0	IND
	-0	0	0	IND	IND	0	0	IND
	+∞	+∞	-∞	+∞	-∞	IND	IND	IND
	-∞	-∞	+∞	-∞	+∞	IND	IND	IND
	±IND	IND	IND	IND	IND	IND	IND	IND

INTEGER ARITHMETIC

Central processor has no 60-bit integer multiply or divide instructions. Integer multiplication and division are performed with 48-bit arguments. The exponent of the result is set to zero. 48-bit integer multiplication is performed with an integer multiply instruction, but integer division must be performed in the floating divide unit. Integer arithmetic is accomplished by putting the integers into unnormalized floating point format using the pack instruction with a zero exponent value.

In integer division, the exponent of the resulting quotient is removed and the result is shifted to compensate for the fact that the result was normalized. In FORTRAN Extended, integer results of multiplication or division are expressed within 48 bits. Full 60-bit one's complement integer sums and differences are possible internally as the central processor has integer addition and subtraction instructions. However, because the binary-to-decimal conversion routines use multiplication and division, the range of integer values output is limited to those which can be expressed with 48 bits.

DOUBLE PRECISION

Although complete arithmetic instructions using double precision arguments are not provided by the hardware, the FORTRAN compiler generates code for true double precision by using instructions which give upper and lower half results with single precision arguments.

COMPLEX

Complex arithmetic instructions are not provided by hardware. The FORTRAN compiler generates code for complex arithmetic by using single precision floating point instructions.

LOGICAL AND MASKING

Logical and masking operations are provided by hardware logical instructions which operate on the entire 60-bit word (refer to section 2, part 1).

ARITHMETIC ERRORS

Under the SCOPE operating system arithmetic errors are classified at execution time as mode 1 - 7:

Mode	Error
1	Address out of range Reference to LCM or SCM outside established limits LCM or SCM block range
2	Operand is an infinite number
3	Address out of range or operand is infinite number
4	Indefinite operand
5	Address out of range or indefinite operand
6	Operand is infinite or indefinite number
7	Operand is infinite, indefinite or address is out of range

Mode 1 — Address out of range. A non-existent storage location has been referenced. Mode 1 errors may be caused by:

calling a non-existent subprogram during execution

using an incorrect number of arguments when calling a subprogram

a subscript assuming an illegal value

no dimensons specified for an array name

Mode 2 — Infinite operand. One of the operands in a real operation is infinite. Infinity is the result whenever the true result of a real operation would be too large for the computer, or when division by zero is attempted. A value of infinity may be returned when some functions are referenced. For example, ALOG(0.) would be negative infinity.

In the following example, Z would be given the value infinity, and when the addition Z + 56. is attempted execution terminates with a mode 2 error.

```
1 FORMAT (F12.3)
  Y = 0.
  Z = 23.2/Y
  PRINT 1, Z
  CAT = Z + 56.
```

When the print statement is executed, an R is printed to indicate an out of range value.

Mode 3 — Address is out of range or operand is infinite number.

Mode 4 — Indefinite operand. One of the operands in a real operation is indefinite. An indefinite result is produced by dividing 0. by 0. or multiplying an infinite operand by 0. An illegal library function reference may return an indefinite value. For example, SQRT (-2.) would produce an indefinite result. An attempt to print an indefinite value produces the letter I.

Mode 5 — Address is out of range or indefinite operand.

Mode 6 — Operand is infinite or indefinite. A mode 6 arithmetic error occurs when a real operation is performed with one operand infinite and the other operand indefinite.

Mode 7 — Operand is infinite, indefinite, or address is out of range.

When an arithmetic error occurs the following type of message appears in the dayfile and execution is terminated:

```
14.39.06.ERROR MODE = 2.   ADDRESS =002135
```

When an arithmetic error occurs, the following type of message appears in the dayfile.

```
14.30.36*00012.059*SYS.        SC006 -      SCM DIRECT RANGE
```

The arithmetic error messages are listed under the following headings:

CODE, MESSAGE AND MEANING, and LEVEL.

CODE xxnnn

xx	SC or JM	The error was issued by System Control (SC) or Job Management (JM). The System Control area of SCOPE provides the control and structure of the operating system. The major portion of the SCOPE operating system is written and loaded as overlays; System Control provides the overlay loaders and some communication between overlays. The Job Management area of SCOPE controls user program input/output, and prepares user programs for execution.
nnn		Index number of the message.

MESSAGE AND MEANING

The message and an interpretation (if necessary) are printed.

LEVEL

Indicates the level of severity of the error as follows:

X	Job terminates. No EXIT processing occurs.
F	Job terminates. EXIT processing occurs.
W	Warning message is printed, and error is ignored. Processing continues, although the portion of the program containing the error may not be executed.
I	Informative message is printed.

CODE	**MESSAGE AND MEANING**	**LEVEL**
SC001	LCM PARITY	F
SC002	SCM PARITY	F
SC003	LCM BLOCK RANGE	F
SC004	SCM BLOCK RANGE	F
SC005	LCM DIRECT RANGE	F
SC006	SCM DIRECT RANGE	F
SC007	PROGRAM RANGE	F
SC008	BREAKPOINT	F
SC009	STEP CONDITION	F
SC010	INDEFINITE CONDITION	F

CODE	MESSAGE AND MEANING	LEVEL
SC011	OVERFLOW CONDITION	F
SC012	UNDERFLOW CONDITION	F
SC040	JOB MAKING 6000 REQUEST IN RAS+1 RAS+1 of user area is non-zero.	F

TRACING ARITHMETIC ERRORS

The following example outlines a method for detecting the location of an arithmetic error. When the following program is executed:

```
      PROGRAM ERR (OUTPUT,TAPE1=OUTPUT)
      NAMELIST /OUT/T,E
      DATA T,E/5.,1./
    1 WRITE (1,OUT)
      E = E/T + 1.
      T = T - 1.
      GO TO 1
      END
```

this message appears in the dayfile:

```
07.11.39.ERROR MODE = 2.  ADDRESS =002146
```

2146 is one plus the address at which the error was detected. The error was detected at address 2145. To locate this address in the program, turn to the Load Map and read the entries under PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE
ERR	100	2062	LGO
NAMOUT=	2162	561	FORTRAN-L
SYSTEM$	2743	664	FORTRAN-L
GETFIT$	3627	32	FORTRAN-L
/OPEN.FO/	3661	6	
OPEN.RM	3667	247	SYSIO-L
/GET.RT/	4136	11	
Z.SQ	4147	77	SYSIO-L
/GET.BT/	4246	5	
BTRT.SQ	4253	130	SYSIO-L

The user program ERR occupies storage locations 100 through 2161, systems program NAMOUT= occupies locations 2162 through 2742, SYSTEM$ 2743 through 3626, GETFIT$ 3627 through 3660, etc. Location 2145 lies between 100 and 2162 and is therefore in the main program ERR. It is location 2045 relative to the beginning of ERR (all locations are relative to the first word address of the program load) 2145 - 100 = 2045 (octal).

This message appears in the dayfile:

```
SYS. SC011 - OVERFLOW CONDITION
```

The address at which the error occurred is given in the dump of the exchange package for the program.

address at which the error occurred

```
XJP DUMP

P   00 000227 A0 061000 B0 000000  SC(A0)=                          SC(P )= 0130 0000 0700 0000 0320
RAS 00 015400 A1 000004 B1 000001  SC(A1)= 2000 0030 0000 0000 0017 SC(B1)= 0000 0000 0000 0000 0000
FLS 00 061000 A2 000005 B2 000000  SC(A2)= 0000 0030 0000 0000 0000 SC(B2)= 0000 0000 0000 0000 0000
PSD 00 060000 A3 000000 B3 000020  SC(A3)= 0000 0030 0000 0000 0000 SC(B3)= 0000 0000 0000 0000 0000
RAL 00 257000 A4 000000 B4 000001  SC(A4)= 0000 0030 0060 0000 0000 SC(B4)= 0030 0000 0000 0000 0000
FLL 00 020000 A5 000126 B5 000111  SC(A5)= 0006 0452 0000 0000 0325 SC(B5)= 0000 0000 0000 0000 0312
NEA 00 015020 A6 000120 B6 000000  SC(A6)= 0211 1600 0000 0000 0000 SC(B6)= 0000 0000 0000 0000 0000
EEA 00 010460 A7 000000 B7 000000  SC(A7)= 0000 0000 0000 0000 0000 SC(B7)= 0000 0000 0000 0000 0000
```

To locate this address in the program turn to the Load Map and read the entries under BLOCK ADDRESS.

```
BLOCK          ADDRESS    LENGTH      FILE

ERR                100      2062      LGO
NAMOUT=           2162       561      FORTRAN-L
SYSTEM$           2743       664      FORTRAN-L
GETFIT$           3627        32      FORTRAN-L
FIX               3661         4      FORTRAN-L
/OPEN.FO/         3665         6
OPEN.PM           3673       247      SYSIO-L
/GET.PT/          4142        11
Z.SO              4153        77      SYSIO-L
/GET.PT/          4252         5
RTPT.SO           4257       130      SYSIO-L
```

The source listing of ERR includes the following code:

```
      PROGRAM ERR (OUTPUT,TAPE1=OUTPUT)
      NAMELIST /OUT/T,E
      DATA T,E/5.,1./
    1 WRITE (1,OUT)
      E = E/T + 1.
      T = T - 1.
      GO TO 1
      END
```

Checking the reason for the error (Arithmetic Error Mode 2 which indicates a real arithmetic error), check the output of the program and the logic of the source program. It becomes apparent that T assumes the value 0. and division by 0. is attempted in statement E=E/T+1. A mode 2 error which signifies an infinite operand can be caused by division by 0. Therefore, the error was caused by statement 06 where division by 0. was attempted.

STRUCTURE OF INPUT/OUTPUT FILES

DEFINITIONS

Record	Data created or processed by: One unformatted WRITE/READ One card image or a print line defined within a formatted WRITE/READ. The slash indicates the end of a record anywhere in the FORMAT specification list. One WRITMS/READMS One BUFFER IN/OUT
Physical record	Data between inter-record gaps; it need not contain a fixed amount of data. A physical record is defined only on magnetic tape.
Physical Record Unit (PRU)	The largest unit of information that can be transferred between a peripheral storage device and central memory/small core storage.
File	A collection of records referenced by one file name.
Logical file	A portion of a file demarcated by FORTRAN ENDFILE statements.

MAXIMUM PHYSICAL RECORD UNIT SIZE

Physical Record on:	Coded	Binary
Disk	640 characters	640 characters
Magnetic tape in SCOPE format	1280 characters	5120 characters
S Tapes	5120 characters	5120 characters
L Tapes	limited only by buffer size	

RECORD MANAGER

The following tables provide brief descriptions of the block/record formats supported by the Record Manager. Detailed information on these formats is available in the Record Manager Reference Manual.

Logical Record Type	Description
F	Fixed length records
D	Record length is given as a character count, in decimal, by a length field contained within the record.
R	Record terminated by a record mark character specified by the user.
T	Record consists of a fixed length header followed by a variable number of fixed length trailers – the header contains a trailer count field in decimal.
U	Record length is defined by the user.
W	Record length is contained in a control word prefixed to the record by the operating system.
Z	Record is terminated by a 12-bit zero byte in the low order byte position of a 60-bit word.
S	One or more physical record units terminated by a short physical record unit.
B	Record length given as a character count in binary by a length field in first four characters of record.

Block Type	Description
K	All blocks contain a fixed number of records; the last block can be shorter.
C	All blocks contain a fixed number of characters; the last block can be shorter.
E	All blocks contain an integral number of records. Block sizes may vary up to a fixed maximum number of characters.
I	A control word is prefixed to each block by the operating system.

The following table specifies combinations of block and record types that can be processed by a FORTRAN program, where x = legal:

Block Type	Record Type								
	F	D	R	T	U†	W	Z	S	B
K	x	x	x	x	x	x	x		x
C	x	x	x	x		x	x	x	x
E	x	x	x	x		x	x		x
I						x			

† Must be blocked one record per block
Buffer statements only

FORTRAN DEFAULT CONVENTIONS (SEQUENTIAL FILES)

File organization = Sequential

Block type = I for unformatted, C for formatted   No blocking

Record type = W for unformatted, Z for formatted

External character code = Display code

Label type = Unlabeled

Maximum block length = 5120 characters

Positioning before first access = No rewind

Positioning of current volume before swap = Unload

Positioning after last access = No rewind

Processing direction = Input/output

Error options = A (accept)

Error options = T (terminate) for READ/WRITE, AD (accept and display) for BUFFER input/output

Suppress multiple buffer = No (Record Manager anticipates user requirements)

Conversion mode = No

A unit record is one W format record. One formatted WRITE can create several unit records. One formatted read can process as input several unit records.

The default values for files named INPUT, OUTPUT and PUNCH are:

Block type C and record type Z.

Buffer input/output files default to C type blocks and S type records.

The appropriate conversion mode is set for all buffer input/output operations.

The conversion mode is set prior to the first open and cannot be changed during the processing of a file.

FORTRAN DEFAULT CONVENTIONS (RANDOM FILES)

When a file is processed using mass storage subroutines, the following file attributes are provided by the FORTRAN compiler:

File organization = Word addressable

Record type = W

Positioning before first access = None Rewind

Positioning after last access = Unchanged Rewind

Processing direction = Input/output

Error options = AD (accept and display)

Suppress multiple buffer = YES (Record Manager does not anticipate file access)

Block type, external character code, label type, maximum block length, positioning of current volume before swap, character conversion and label creation/checking are not applicable.

One WRITMS creates one W format record. One READMS reads one W format record. The master index is the last W format record in the file. If the length specified for a READMS is longer than the record, the excess locations in the user area are not changed by the read. If the record is longer than the length specified for a READMS, the excess words in the record are skipped.

ADDITIONAL BLOCK AND RECORD TYPES

FILE Control Cards: The FORTRAN programmer can use the Record Manager FILE control card to override the default values supplied by the FORTRAN compiler. This control card is described in 6000 Record Manager Reference Manual. The 7600 programmer can use the FILE control card to access record types other than W. This control card is described in 7600 SCOPE Reference Manual. Section 6, part 3 describes the FORTRAN/Record Manager interface.

BLOCK TYPES

Any block type consistent with the record type can be specified in the FILE control card because the FORTRAN language does not contain any statements which specify or constrain the block type.

RECORD TYPE

Although FORTRAN does not contain statements which specify the record type, constraints are imposed on the processing of certain types of records because of the logical structure of the records. When specifying record types in the FILE control card, the FORTRAN programmer must be aware of the following constraints:

Record Type	Action Required
D	User must insert the record length on write. The length field must have leading zeros, not blanks, and must be within the buffer length used by the FORTRAN object time input/output routines.
R	User must insert the record mark on write.
B	User must insert the record length on write.
T	User must insert the trailer count field; it must have leading zeros not blanks. The same buffer restrictions apply as for D format records.
F	User must ensure fixed length records. Records larger than the fixed length are truncated; shorter records are padded with blanks.
Z	User must ensure that the character configuration used as a line terminator (two colons) does not occur at the low order byte position of a 60-bit word.
	User must also ensure recording mode is set to unformatted for tape.
U	User must ensure that only one record per block is written. Only block type K, one record per block, is allowed with U type records.
	BUFFER statements only.
S	User must be aware that each record (as defined on page 5-1) may occupy several physical records.
	User must be aware that each record is an S record.

This deck illustrates the use of the FILE card to override default values supplied by the FORTRAN compiler. The following values are used:

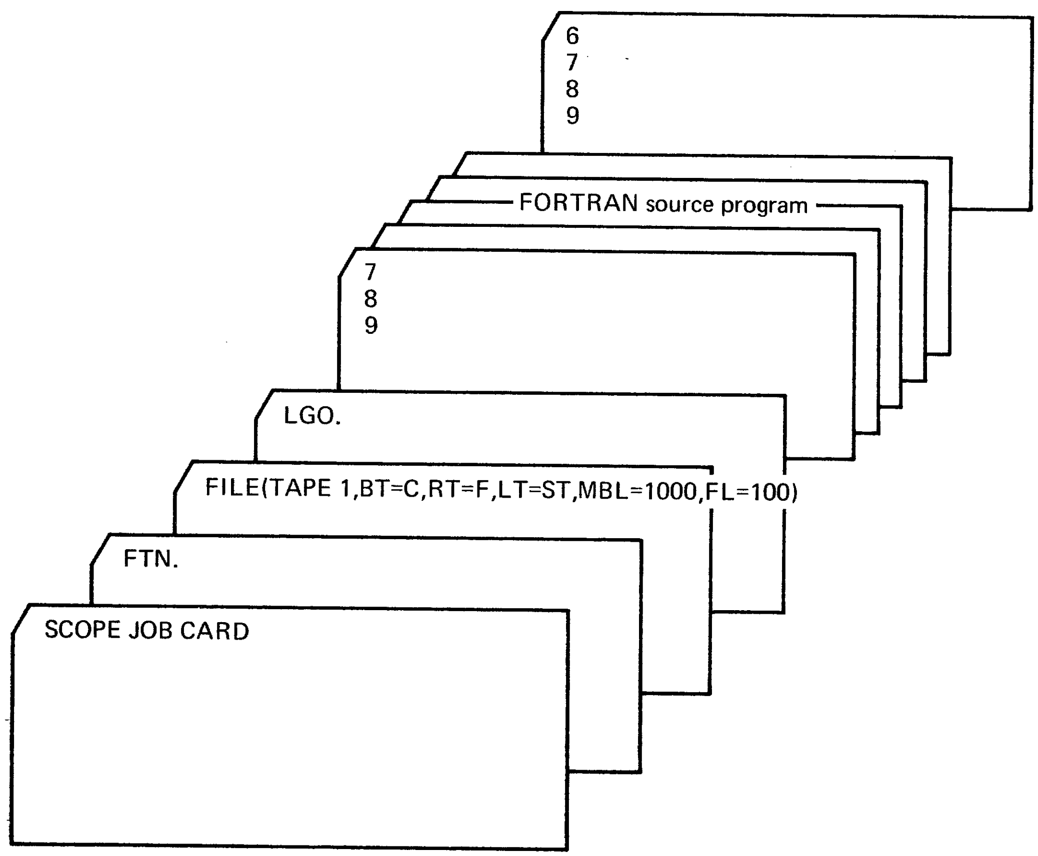
Label Type = standard labels (LT=ST)

Block Type = character count (BT=C)

Maximum Block Length = 1000 characters (MBL=1000)

Record Type = fixed length (RT=F)

Record Length = 100 characters (FL=100)



This deck illustrates the use of the FILE card to override default values supplied by the FORTRAN compiler. The following values are used:

Label Type = unlabeled

Block type = character count

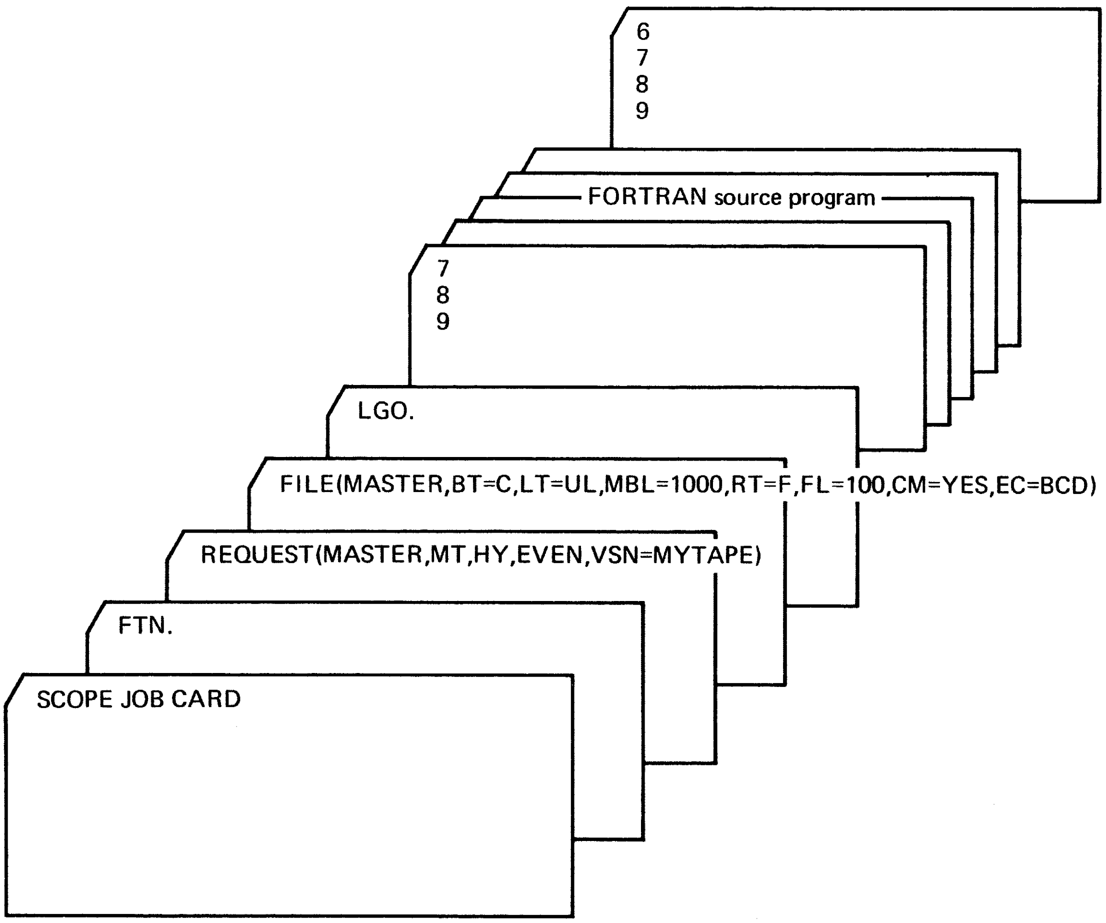
Maximum block length = 1000 characters

Record type = fixed length

Record length = 100 characters

Conversion mode = YES

External code = BCD



Assuming the source program is using formatted writes and 100-character records are always written, the file will be written on magnetic tape in 1000-character blocks (except possibly the last block) with even parity, at 800 bpi. No labels will be recorded, and no information will be written other than that supplied by the user. Records will be blocked 10 to a block.

BACKSPACE/REWIND

Backspacing on FORTRAN files repositions them so that the last logical record becomes the next logical record.

BACKSPACE is permitted only for files with F, S, or W record format or tape files with one record per block.

The user should remember that formatted input/output operations can read/write more than one record; unformatted input/output and BUFFER IN/OUT read/write only one record.

The rewind operation positions a magnetic tape file such that the next FORTRAN input/output operation references the first record. A mass storage file is positioned to the beginning of information.

The following table details the actions performed prior to positioning.

BACKSPACE/REWIND

Condition	Device Type	Action
Last operation was WRITE or BUFFER OUT	Mass Storage	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written.
	Unlabeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written. Two file marks are written.
	Labeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted record is written. A file mark is written. A single EOF label is written. Two file marks are written.

Condition	Device Type	Action
Last operation was WRITE. BUFFER OUT or ENDFILE	Mass storage (no blocking)	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written.
	Unlabeled Magnetic Tape or Blocked Mass Storage	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written. Two file marks are written (on tape).
	Labeled Magnetic Tape or Labeled Blocked Mass Storage	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written. A file mark is written. A single EOF label is written. Two file marks are written.
Last operation was READ, BUFFER IN or BACKSPACE	Mass Storage	None
	Unlabeled Magnetic Tape	None
	Labeled Magnetic Tape	If the end of information has been reached, labels are processed.
No previous operation	Magnetic Tape	If the file is assigned to on-line magnetic tape, a REWIND request is executed. If the file is staged, the REWIND request has no effect. The file is staged and rewound when it is first referenced.
	Mass Storage	REWIND request causes the file to be rewound when first referenced.
Previous operation was REWIND		Current REWIND is ignored.

ENDFILE

The ENDFILE operation introduces a delimiter into an input/output file. The following table shows the effect of ENDFILE on various record types.

Record Type	Action
W	Write end-of-partition. Terminate current block for magnetic tape file.
S	Terminate current block for magnetic tape file. Write level 17 zero length block.
Z with C blocking	Terminate current block for magnetic tape file. Write level 17 zero length block.
D,B,R,T	
F,U, or Z	Terminate current block for magnetic tape file. Write level 17 zero length block.

A WRITE/BUFFER OUT can follow an ENDFILE operation. If the file has records of the format W,S, or Z with C blocking or it is a mass storage file with any other block/record formats, no special action is performed. However, if the file is assigned to magnetic tape and has a record format other than W, S, or Z with C blocking a tape mark is written preceding the requested record.

LABELED FILES

Only files recorded on magnetic tape can be labeled files.

To process labeled files through a FORTRAN program, the SCOPE LABEL statement (refer to SCOPE Reference Manual) must specify a value for the label creation/checking parameter: either R which specifies an existing label is to be checked, or W which creates labels if no labels exist. The processing direction for sequential files in FORTRAN must be input/output to permit both READ and WRITE by a FORTRAN program on the file; therefore, the user must specify whether a labeled magnetic tape is used for input, or output/input/output.

If a file is declared to be labeled on a REQUEST control card, SCOPE compares the label (HDR1 only) with the label expected by the user. If the information does not compare and the use of the file is input, the job continues after instructions are entered from the system console. For output, a default label is written, and the job continues.

An object time subroutine, LABEL, is provided for the FORTRAN programmer to set up label information for Record Manager. If label information is properly set up, and subroutine LABEL is referenced prior to any other reference to the file, 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.

Form of the call:

```
      7
      CALL LABEL (u,fwa)
```

u — Unit number

fwa — Address of first word containing the label information which must be in the mode and format discussed in the SCOPE Reference Manual.

Alternatively, the LABEL utility can be used (refer to SCOPE Reference Manual).

BUFFER INPUT/OUTPUT

The maximum lengths for physical records on tape can be exceeded using the BUFFER input/output statements if the L parameter on the SCOPE REQUEST control card is specified.

BUFFER IN/OUT statements can be used to achieve some degree of overlap between the user program and input/output with an external device (mass storage or tape). However, the memory area specified in the BUFFER IN/OUT statement will not be used as the physical record buffer. These buffers are maintained within an operating system buffer area in LCM. The execution of a BUFFER IN/OUT statement, therefore, involves movement of a record between system buffers in LCM and the memory area specified in the BUFFER IN/OUT statement. Correspondence between individual BUFFER statements and physical records on a device depends upon the block specification. For example, K blocking with a record count of one ensures that each BUFFER IN/OUT corresponds to a block.

BUFFER IN

1. Only one record is read each time a BUFFER IN is performed. If the length specified by the BUFFER statement is longer than the record read, excess locations are not changed by the read. If the record read is longer than the length specified by the BUFFER statement, the excess words in the record are ignored. The number of central memory words transmitted to the program block can be obtained by referencing the function LENGTH (section 8, part 1).

2. After using a BUFFER IN/OUT statement on unit u, and prior to referencing unit u or the contents of storage locations a through b, the status of the BUFFER operation must be checked by a reference to the UNIT function (section 8, part 1). This status check ensures that the data has actually been transferred, and the buffer parameters for the file have been restored.

3. If an attempt is made to BUFFER IN past an end-of-file without testing for the condition by referencing the UNIT function, the program terminates with the diagnostic:

   END OF FILE ENCOUNTERED file name

4. If the last operation on the file was a write operation, no data is available to read. If a read is attempted, the program terminates with the diagnostic:

   WRITE FOLLOWED BY READ ON FILE

5. If the starting address for the block is greater than the terminal address, the program terminates with the diagnostic:

   BUFFER DESIGNATION BAD FWA.GT.LWA, file name

6. If an attempt is made to BUFFER IN from an undefined file (a file not declared on the PROGRAM card), the program terminates with the diagnostic:

   UNASSIGNED MEDIUM, file name

BUFFER OUT

1. One record is written each time a BUFFER OUT is performed. The length of the record is the terminal address of the record (LWA) – starting address (FWA) + 1.

2. As with BUFFER IN, a BUFFER OUT operation must be followed by a reference to the UNIT function. This reference must occur prior to any other reference to the file.

3. If the terminal address is less than the first word address, the program terminates and the following diagnostic is issued:

   BUFFER SPECIFICATION BAD FWA.GT.LWA, file name

4. The UNASSIGNED MEDIUM diagnostic is similar to that issued from a BUFFER IN.

STATUS CHECKING

UNIT FUNCTION (BUFFERED INPUT/OUTPUT)

The UNIT function is used to check the status of a BUFFER IN or BUFFER OUT operation on logical unit u. The function returns the following values:

-1 Unit ready, no error

+0 Unit ready, end-of-file encountered on the previous operation

+1 Unit ready, parity error encountered on the previous operation

Example:

```
IF (UNIT(5)) 12,14,16
```

Control transfers to the statement labeled 12, 14 or 16 if the value returned was -1., 0., or +1. respectively.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. If the UNIT function references a logical unit referenced by input/output statements other than BUFFER IN/BUFFER OUT, the status returned will always indicate unit ready and no error (-1.).

Any of the following conditions encountered during a read result in end-of-file status:

End of information

Non-deleted W format flag record

Embedded tape mark

Terminating double tape mark

Terminating end of file label

Embedded zero length level 17 block

At end of section on INPUT file only

EOF FUNCTION (NON-BUFFERED, INPUT/OUTPUT)

The EOF function is used to test for an end-of-file read on unit u. Zero is returned if no end-of-file is encountered, or a non-zero value if end-of-file is encountered.

Example:

```
IF (EOF(5)) 10,20
```

returns control to the statement labeled 10 if the previous read encountered an end-of-file, otherwise control goes to statement 20.

If an end-of-file is encountered, EOF clears the indicator before returning control.

If the previous operation on unit u was a write, EOF will return a zero value. An end-of-file condition exists only when an end-of-file is read.

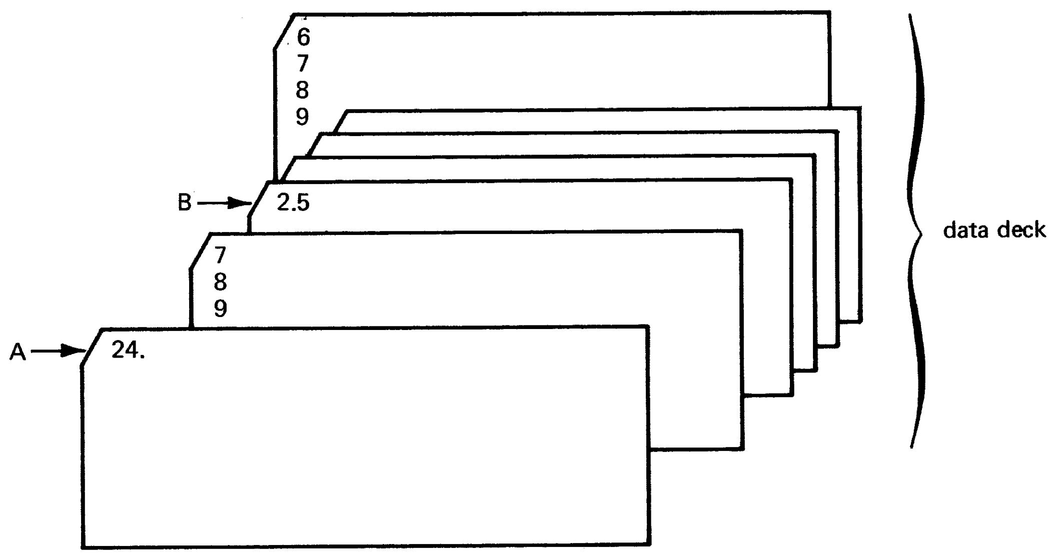
This function has no meaning when applied to a random access file. If the EOF function is called in reference to such a file, a zero value would be returned.

Refer to the UNIT function for a list of conditions which result in an end-of-file status.

The user should test for an end-of-file after each READ statement to avoid input/output errors. If an attempt is made to read on unit u and an EOF was encountered on the previous read operation on this unit, execution terminates and an error message is printed.

Example:

If the values in storage are A = 10., B = 44., and C = 3., and an EOF is reached after A is read, B and C are not read nor altered in memory.




```
      READ 1, A,B,C
1     FORMAT (F4.2)
```

After the read statement, A will contain 24., B = 44. and C = 3. The end-of-file flag will be set. The job will be terminated if a subsequent READ is attempted without first executing an EOF check.

IOCHEC FUNCTION

The IOCHEC function tests for parity errors on non-buffered reads on unit u. The value zero is returned if no error occurs.

Example:

```
J = IOCHEC(6)
IF (J) 15,25
```

zero value would be returned to J if no parity error occurred and non-zero if an error had occurred; control would transfer to the statement labeled 25 or 15 respectively.

If a parity error occurred, IOCHEC would clear the parity indicator before returning. Parity errors are handled in this way regardless of the type of the external device.

PARITY ERROR DETECTION

A parity error status indicates that a parity error occurred within the current record. For non-buffered formatted files. the error did not necessarily occur within the last record requested by the program because the input/output routines read ahead one record whenever possible.

Parity errors are detected by the status checking functions on all read operations. When the write check option is specified on the REQUEST statement for the file (7600 SCOPE V2.0 Reference Manual) parity status may be checked on write operations which access mass storage files. Write parity errors for other types of devices (staged/on-line tape) are detected by the operating system, and a message is written in the dayfile.

When parity error status is returned, this does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the Record Manager input/output routines via buffers in large core memory.

DATA INPUT ERROR CONTROL

The subprogram ERRSET allows a complete data set to be processed in one pass without premature termination due to errors; a listing is produced of all data errors and input diagnostics.

ERRSET (a,b) is called before a READ statement; it initializes an error count cell, a, and establishes a maximum number of errors, b. The program does not terminate when fatal errors are encountered until the limit,b, is reached. A maximum limit of $2^{59}-1$ can be specified.

The limit continues in effect for any subsequent READ statements until the number of errors specified has accumulated. The limit can be reset before a READ statement or turned off by setting b=0; b=0 is the equivalent of a normal read.

Example:

The following example illustrates the use of ERRSET to suppress normal fatal termination when large sets of data are being processed.

When ERRSET is called, a limit of 200 errors is established. The number of errors will be stored in KOUNT. After ARAY is read, KOUNT is checked. If errors occur, the following statements are not processed and a branch is made to statement 500. Had ERRSET not been called, fatal errors would have terminated the program before the branch to statement 500. At statement 500, ERRSET once more initializes the error count and compilation continues.

```
    CALL ERRSET(KOUNT,200)
    READ(1,125)(ARAY(I),I=1,1500)
125 FORMAT (3F10.5,E10.1)
    IF (KOUNT.GT.0) GO TO 500
         .
         .
         .
500 CALL ERRSET(KOUNT,200)
    READ(1,125)(BRAY(I),I=1,1500)
    IF (KOUNT.GT.0) GO TO 600
         .
         .
         .
600 CALL ERRSET(KOUNT,100)
    READ(1,230)(LRAY(I),I=1,500)
    PRINT 99, KOUNT
    READ(4,127)(MRAY(I),I=1,500)
    PRINT 99, KOUNT
    READ(4,225)(NRAY(I),I=1,50)
         .
         .
         .
         .
         .
    IF (KOUNT.GT.0) GO TO 700
         .
         .
         .
700 CALL EXIT
    END
```

Data errors and diagnostics are listed, providing the programmer with a list of errors for the entire program:

ERROR MESSAGES

```
ERROR IN COMPUTED GOTO STATEMENT- INDEX VALUE INVALID
ERROR NUMBER   0001 DETECTED BY ACGOER  AT ADDRESS 000001
CALLED FROM EXERR   AT  LINE 0003
```

DIAGNOSTICS

1. Illegal Data in Field

```
*ERROR DATA INPUT* ILLEGAL DATA IN FIELD
**FORMAT NO. 125
```

2. Data overflow, exponent subfield has exceeded 323 (decimal) (data underflow, exponent less than -323.)

```
*ERROR DATA INPUT* DATA OVERFLOW
** FORMAT NO. 125
```

3. Both illegal data and data overflow have been detected.

```
*ERROR DATA INPUT* ILLEGAL DATA IN FIELD **
AND DATA OVERFLOW ** FORMAT NO. 125
```

An error summary appears at the end of the program.

Error Summary:

ERROR	TIMES
0078	0003
0079	0001

PROGRAMMING NOTES

Meaningful results are not guaranteed in the following circumstances:

1. Mixed binary and display code files. For example, formatted and unformatted READ should not be mixed.

2. Mixed buffer input/output statements and 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 input/output statements on the same file without the intervening execution of a UNIT function call.

The FORTRAN user communicates with Record Manager through two types of FORTRAN CALL statements: calls that create, access, and modify the file information table and file commands that position and process files.

FILE INFORMATION TABLE CALLS

To place values in the file information table the user can call one of the following subroutines:

FILESQ for sequential files

FILEWA for word addressable files

FILEIS for indexed sequential files

FILEDA for direct access files

```
      7
      CALL FILExx (fit,keyword1,value1,. . . ,keywordn,valuen)
```

xx — SQ, WA, IS, or DA

fit — Name of an array. Record Manager resides in the user's field length, and the array must be large enough to contain both the file information table (FIT) and the file environment table (FET). 35 words should be allocated; 20 words for the file information table and 15 words for the file environment table. The file information table is created by the subroutine FILExx, beginning in the first word of the array. Record Manager supplies the information for the SCOPE file environment table, which is placed in the user's array after the file information table.

keyword — Specifies a file information table field.

value — Value to be placed in the file information table field specified by the keyword.

All parameters, with the exception of fit, are paired; the first parameter is the keyword which indicates the field in the file information table, the second parameter is the value to be placed in the field. Only the pertinent parameters need be specified, and they may appear in any order. Since a FORTRAN call can contain a maximum of 63 parameters, 31 file information table fields can be specified with a FILExx call.

Example:

The following call sets up a file information table for a direct access file:

```
CALL FILEDA (FILE,3LLFN,7LSDAFILE,3LFWB,BUFFER,3LBFS,400,3LBCK,3LYES)
```

The file information table and the file environment table are to be constructed in the array named FILE. The file name (LFN) is SDAFILE. The buffer is to be placed in a 400-word array BUFFER. 3LBCK,3LYES selects the block checksumming option.

Keywords and symbolic options, such as YES and NO, are passed as L format Hollerith constants.

For example:

3LFWB 3LBCK	keywords
3LYES 2LNO	symbolic options

Numeric parameters must be integer variables or integer constants.

UPDATING FILE INFORMATION TABLE

After the file information table is created, it can be updated by calls to the subroutine STOREF.

```
      7
      CALL STOREF (fit,keyword,value)
```

fit	Array where the file information table was created.
keyword	File information table field.
value	Value to be placed in the field.

Example:

```
CALL STOREF (FILE,2LRL,250)
```

Sets record length in the array FILE to 250 characters.

Contents of file information table fields can be accessed by using the integer function IFETCH.

IFETCH (fit,keyword)

fit	Name of the array containing file information table.
keyword	Name of the field.

If the keyword is a one-bit field, the result is returned in the sign bit and can be sensed by a positive-negative check; otherwise, it is returned right justified with zero fill.

Example:

```
M=IFETCH (FILE,2LRL)
```

The record length is returned to the function IFETCH and replaces the value of M.

FILE COMMANDS

After the file information table is created, file processing commands can be issued. (FORTRAN file processing commands correspond to Record Manager macros of the same name.)

The first command must be OPENM, and the last CLOSEM. File command calls have positional parameters. If trailing parameters are omitted, the current value in the file information table will be used. However, no other parameters may be omitted; if the current value in the file information table is to be used, it must be specified. For example:

```
CALL PUT (fit,wsa,rl,ka/wa,kp,pos,ex)
```

If rl and the following parameters are to be specified, the parameter wsa cannot be omitted even if its value does not change, but ex could be omitted if the current value is to be used.

In the following calls, when two parameters occur in one position, for example ka/wa in CALL PUT, the first parameter applies to indexed sequential or direct access files, and the second to word addressable or sequential files.

In the following description of the file commands, fit is the name of an array containing the file information table.

```
      CALL OPENM (fit,pd,of)
```

OPENM prepares a file for processing. Each file must be opened before processing.

pd Processing direction established when file is opened:

INPUT	Read only
OUTPUT	Write only
I-O	Read and write
NEW	Indexed sequential or direct access file to be created (write only)

of Open flag specifies position of file when it is opened:

R	Rewind; file is rewound before any other open procedures are performed.
N	No file positioning is done before other open procedures.
E	File is positioned immediately before end of information to allow extensions to a mass storage file.

```
      CALL CLOSEM (fit)
```

Terminates processing.

```
      CALL GET (fit,wsa,ka/wa,kp,mkl,rl,ex/dx)
```

GET reads a record from an input/output device and delivers it to the user's record area.

wsa	Address of user's record area.
ka	Address of user's key area for direct access or indexed sequential record to be read.
wa	Word address on file where reading is to start.

kp — Beginning character position of key within ka. Key positions are ordered from left to right (0-9).

mkl — Major key length on indexed sequential files.

ex — Address of exit routine to be entered when an error occurs (word addressable, indexed sequential or direct access files). The value of ex must not be zero.

dx — Address of end of data routine for sequential files.

rl — Record length in characters.

```
      7
      CALL PUT (fit,wsa,rl,ka/wa,kp,pos,ex)
```

PUT places a record in a file.

pos — Specifies relative position of record for duplicate key processing.

wsa,rl,ka,wa,kp,ex are the same as for GET.

```
      7
      CALL GETN (fit,wsa,ka,ex)
```

GETN accesses the next record on the file.

```
      7
      CALL DLTE (fit,ka/wa,kp,pos,ex)
```

DLTE removes a record from a file.

wa — Word address of record to be deleted.

pos — Specifies the last referenced (current) record will be deleted. Applies only when duplicate key processing is allowed.

```
      7
      CALL REPLC (fit,wsa,rl,ka,kp,pos,ex)
```

REPLC replaces an existing record with a record from the user's record area.

pos — Specifies the last referenced (current) record will be replaced. Applies only when duplicate key processing is allowed.

```
      7
      CALL CHECK (fit)
```

CHECK determines whether input/output operations on a file are complete and upon completion returns control

```
      7
      CALL SKIP (fit,±count)
```

Repositions a file.

count — Number of logical records to be skipped; positive for a forward skip, negative for a backward skip.

```
      7
      CALL SEEKF (fit,ka,kp,mkl,ex)
```

SEEKF allows central memory processing to overlap input/output operations.

```
      7
      CALL WEOR (fit,lev)
```

WEOR terminates a section, and an S type record.

lev — Level number (any octal value 0 to 16) to be appended if record type is S; default is 00.

```
      CALL WTMK (fit)
```

Writes a tape-mark.

```
      CALL ENDFILE (fit)
```

Writes an end of partition.

```
      CALL REWND (fit)
```

RWND positions a tape file to the beginning of the current volume. It positions a mass storage file to the beginning of information.

```
      CALL GETP (fit,wsa,ptl,4LSKIP,dx)
```

GETP retrieves partial records; it may be used to retrieve an arbitrary amount of data from a record.

wsa	Address of user's record area to receive the record.
ptl	Partial transfer length. Number of characters to be transferred.
4LSKIP	Record Manager advances to next record before retrieving data.
dx	Address of end-of-data routine.

```
      7
      CALL PUTP (fit,wsa,ptl,rl,ex)
```

Writes a portion of a record.

wsa	Address of user's record area from which the record portion will be taken.
ptl	Same as GETP.
rl	Record length in characters (required only for U, W, and R type records).
ex	Address of error routine.

ERROR CHECKING

FORTRAN/Record Manager routines perform limited error checking to determine whether the call can be interpreted, but actual parameter values are not checked.

The following error conditions are detected, and a message appears in the dayfile:

FIT ADDRESS NOT SPECIFIED	Array name was not specified.
FORMAT ERROR	Parameters were not paired (FILExx), or required parameters were not specified (STOREF, IFETCH or SKIP).
UNDEFINED SYMBOL	A file information table field mnemonic or symbolic option was specified incorrectly; for example, an incorrect spelling, or the of parameter in OPENM was not specified as R, N or E.

Example of error message:

```
ERROR IN STOREF CALL

UNDEFINED SYMBOL  IMPUT
```

MASS STORAGE INPUT/OUTPUT III-7

Mass storage input/output subroutines allow the user to create, access, and modify multi-record files on a random basis without regard for their physical position or internal structure. A random file can reside on any mass storage device for which Record Manager word addressable file organization is defined. Each record in the file may be read or written at random without logically affecting the remaining file contents. The length and content of each record is determined by the user.

Six object time input/output subroutines control the transfer of records between central memory and mass storage. These routines employ the word addressable feature available through Record Manager (refer to Record Manager Reference Manual and 7000 SCOPE Reference Manual for details of this feature).

```
      7
      CALL OPENMS (u,ix,lngth,t)
```

OPENMS opens the mass storage file and informs Record Manager that it is a random (word addressable) file. The array specified in the call arguments is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

u	Unit designator
ix	First word address in central memory of the array which will contain the index
lngth	Length of index
	for a number index, lngth ≥ (number of records in file) + 1
	for a name index, lngth ≥ 2*(number of records in file) + 1
t	t = 0 file is referenced through a number master index
	t = 1 file is referenced through a name master index

Example:

```
DIMENSION I(11)
CALL OPENMS (5,I,11,0)
```

Prepares for random input/output on unit 5 with an 11-word master index of the number type. If the file already exists, the master index is read into memory starting at address I.

```
      7
CALL READMS (u,fwa,n,k)
```

Transmits data from mass storage to central memory.

u	Unit designator
fwa	Address in central memory of first word of record
n	Number of 60-bit central memory words in the record to be transferred
k	Number index: k $= 1 \leq k \leq$ lngth - 1
	Name index: k = any 60-bit quantity except ± 0

Example:

```
CALL READMS (3,DATAMOR,25,2)
```

```
      7
CALL WRITMS (u,fwa,n,k,r,s)
```

Transmits data from central memory to the selected mass storage device.

u, fwa, n, k are the same as for READMS.

r	r=1 rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length.
	r=-1 rewrite in place if space available, otherwise write at end of information
	r=0 no rewrite; write normally at end of information

The default value for r is 0 (normal write). The r parameter can be omitted if the s parameter is omitted.

s	s=1 write sub-index marker flag in index control word for this record
	s=0 do not write sub-index marker flag in index control word for this record

Default value is 0 if s is omitted.

The s parameter is included for future random file editing routines. Current routines do not test the flag, but the user should include this parameter in new programs when appropriate to facilitate transition to a future edit capability.

Example:

```
CALL WRITMS (3,DATA,25,6,1)
```

```
      7
CALL STINDX (u,ix,lngth,t)
```

STINDX selects a different array to be used as the current index to the file. The call permits a file to be manipulated with more than one index. For example, when the user wishes to use a sub-index instead of the master index, he calls STINDX to select the sub-index as the current index. The STINDX call does not cause the sub-index to be read or written; that task must be carried out by explicit READMS or WRITMS calls. It merely updates the internal description of the current index to the file.

u, ix, lngth and t are the same as OPENMS.

Examples:

```
DIMENSION SUBIX (10)
CALL STINDX (3,SUBIX,10,0)

DIMENSION MASTER (5)
CALL STINDX (2,MASTER,5)
```

```
      7
CALL CLOSMS (u)
```

The CLOSMS call is optional since its function is identical to that performed automatically by the FORTRAN object time routine SYSTEM when the run terminates. (SYSTEM and CLOSMS both write the master index from central memory to the file, and close the file.) CLOSMS is provided so that a file can be returned to the operating system before the end of a FORTRAN run, or to preserve a file created by an experimental job that may subsequently abort, or for other special circumstances.

Example:

```
CALL CLOSMS (2)
```

ACCESSING A RANDOM FILE

Random file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written, and can normally be read back only in the same order. This can be slow and inconvenient in applications where the order of writing and retrieving records differ and, in addition, it requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, a randomly-accessible file capability is provided by the mass storage input/output subroutines.

In a random file, any record may be read, written or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random-access rotating mass storage device that can be positioned to any portion of a file. Thus, the entire concept of file position does not apply to a random file. The notion of rewinding a random file is, for instance, without meaning.

To permit random accessing, each record in a random file is uniquely and permanently identified by a record key. A key is an 18- or 60-bit quantity, selected by the user and included as a READMS or WRITMS call parameter. When a record is first written, the key in the WRITMS call becomes the permanent identifier for that record. The record can be retrieved later by a READMS call that includes the same key, and it can be updated by a WRITMS call with the same key.

When a random file is in active use, the record key information is kept in an array in the user's field length. The user is responsible for allocating the array space by a DIMENSION, type or similar array declaration statement, but must not attempt to manipulate the array contents. The array becomes the directory or index to the file contents. In addition to the key data, it contains the word address and length of each record in the file. The index is the logical link that enables the mass storage subroutines, in conjunction with Record Manager, to associate a user call key with the hardware address of the required record.

The index is maintained automatically by the mass storage subroutines. The user must not alter the contents of the array containing the index in any manner; to do so may result in destruction of the file contents. (In the case of a sub-index, the user must clear the array before using it as a sub-index; and read the sub-index into the array if an existing file is being reopened and manipulated. However, individual index entries should not be altered.)

In response to an OPENMS call, the mass storage subroutines automatically clear the assigned index array. If an existing file is being reopened, the mass storage subroutines will locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to the mass storage device. When the file is reopened, by the same job or another job, the index is again read into the index array space provided, so that file manipulation may continue.

INDEX KEY TYPES

There are two types of index key, name and number. A name key may be any 60-bit quantity except +0 or -0. A number key must be a simple positive integer, greater than 0 and less than or equal to (lngth - 1). The user selects the type of key by the (t) parameter. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. If the user should inadvertently attempt to reopen an existing file with an incorrect index type parameter, the job will be aborted. (This does not apply to sub-indexes chosen by STINDX calls, proper index type specification is the sole reponsibility of the user.) In addition, key types cannot be mixed within a file. Violation of this restriction may result in destruction of a file.

The choice between name and number keys is left entirely to the user. The nature of the application may clearly dictate one type or the other. However, where possible, the number key type is preferable. Job execution will be faster and less central memory space will be required. Faster execution occurs because it is not necessary to search the index for a matching key entry (as is necessary when a name key is used). Space is saved due to the smaller index array length requirement.

Example:

```
      PROGRAM MS1 (TAPE3)

C  CREATE RANDOM FILE WITH NUMBER INDEX.

      DIMENSION INDEX(11), DATA(25)
      CALL OPENMS (3,INDEX,11,0)

      DO 99 NRKEY=1,10
C               .
C               .
C  (GENERATE RECORD IN ARRAY NAMED DATA.)
C               .
C               .
 99   CALL WRITMS (3,DATA,25,NRKEY)

      STOP
      END



      PROGRAM MS2 (TAPE3)

C  MODIFY RANDOM FILE CREATED BY PROGRAM MS1.
C  NOTE LARGER INDEX BUFFER TO ACCOMMODATE TWO NEW
C  RECORDS.

      DIMENSION INDEX(13), DATA(25), DATAMOR(40)
      CALL OPENMS (3,INDEX,13,0)

C  READ 8TH RECORD FROM FILE TAPE3.
      CALL READMS (3,DATA,25,8)
C               .
C               .
C  (MODIFY ARRAY NAMED DATA.)
C               .
C               .

C  WRITE MODIFIED ARRAY AS RECORD 8 AT END OF
C  INFORMATION IN THE FILE
      CALL WRITMS (3,DATA,25,8)

C  READ 6TH RECORD.
      CALL READMS (3,DATA,25,6)
C               .
C               .
C  (MODIFY ARRAY.)
C
```

```
C

C  REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6.
      CALL WRITMS (3,DATA,25,6,1)

C  READ 2ND RECORD INTO LONGER ARRAY AREA.
      CALL READMS (3,DATAMOR,25,2)
C               .
C               .
C  (ADD 15 NEW WORDS TO THE ARRAY NAMED DATAMOR.)
C               .
C               .

C  CALL FOR IN-PLACE REWRITE OF RECORD 2.  IT WILL
C  DEFAULT TO A NORMAL WRITE AT END-OF-INFORMATION
C  SINCE THE NEW RECORD IS LONGER THAN THE OLD ONE,
C  AND FILE SPACE IS THEREFORE UNAVAILABLE.
      CALL WRITMS (3,DATAMOR,40,2,-1)

C  READ THE 4TH AND 5TH RECORDS.
      CALL READMS (3,DATA,25,4)
      CALL READMS (3,DATAMOR,25,5)
C               .
C               .
C  (MODIFY THE ARRAYS NAMED DATA AND DATAMOR.)
C               .
C               .

C  WRITE THE ARRAYS TO THE FILE AS TWO NEW RECORDS.
      CALL WRITMS (3,DATA,25,11)
      CALL WRITMS (3,DATAMOR,25,12)

      STOP
      END



      PROGRAM MS3 (TAPE7)

C  CREATE A RANDOM FILE WITH NAME INDEX.

      DIMENSION INDEX(9), ARRAY(15,4)
      DATA REC1,REC2/7HRECORD,≠RECORD2≠/
C               .
C               .
C  (GENERATE DATA IN ARRAY AREA.)
C               .
C               .
```

```
C   WRITE FOUR RECORDS TO THE FILE.  NOTE THAT
C   KEY NAMES ARE RECORD(N).
        CALL WRITMS (7,ARRAY(1,1),15,REC1)
        CALL WRITMS (7,ARRAY(1,2),15,REC2)
        CALL WRITMS (7,ARRAY(1,3),15,7RRECORD3)
        CALL WRITMS (7,ARRAY(1,4),15,≠RECORD4≠)

C   CLOSE THE FILE.

        CALL CLOSMS (7)

        STOP
        END
```

MULTI-LEVEL FILE INDEXING

When a file is opened by an OPENMS call, the mass storage routines clear the array specified as the index area, and if the call is to an existing file, locates the file index and reads it into the array. This creates the initial or master index.

The user can create additional indexes (sub-indexes) by allocating additional index array areas, preparing the area for use as described below, and calling the STINDX subroutine to indicate to the mass storage routine the location, length and type of the sub-index array. This process may be chained as many times as required, limited only by the amount of central memory space available. (Each active sub-index requires an index array area.) The mass storage routine uses the sub-index just as it uses the master index; no distinction is made.

A separate array space must be declared for each sub-index that will be in active use. Inactive sub-indexes may, of course, be stored in the random file as additional data records.

The sub-index is read from and written to the file by the standard READMS and WRITMS calls, since it is indistinguishable from any other data record. Although the master index array area is cleared by OPENMS when the file is opened, STINDX does not clear the sub-index array area. The user must clear the sub-index array to zeros. If an existing file is being manipulated and the sub-index already exists on the file, the user must read the sub-index from the file into the sub-index array by a call to READMS before STINDX is called. STINDX then informs the mass storage routine to use this sub-index as the current index. The first WRITMS to an existing file using a sub-index must be preceded by a call to STINDX to inform the mass storage routine where to place the index control word entry before the write takes place.

If the user wishes to retain the sub-index, it must be written to the file after the current index designation has been changed back to the master index, or a higher level sub-index by a call to STINDX.†

---

†Since the file is closed automatically at job termination, it is no longer necessary as it was under previous versions of FORTRAN Extended, for the user to reset the master index before closing the file.

INDEX TYPE

MASTER INDEX

The master index type for a given file is selected by the t parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created; attempts to do so by reopening the file with the opposite type index are treated as fatal errors.

SUB-INDEX

The sub-index type can be specified independently for each sub-index. A different sub-index name/number type can be specified by including the t parameter in the STINDX call. If t is omitted, the index type remains the same as the current index. Intervening calls which omit the t parameter do not change the most recent explicit type specification. The type remains in effect until changed by another STINDX call.

STINDX cannot change the type of an index which already exists on a file. The user must ensure that the t parameter in a call to an existing index agrees with the type of the index in the file. Correct sub-index type specification is the responsibility of the user; no error message is issued.

Example:

```
      PROGRAM MS4 (TAPE2)

C  GENERATE SUBINDEXED FILE WITH NUMBER INDEX.  FOUR
C  SUBINDEXES WILL BE USED, WITH NINE DATA RECORDS
C  PER SUBINDEX, FOR A TOTAL OF 36 RECORDS.

      DIMENSION MASTER(5), SUBIX(10), RECORD(50)
      CALL OPENMS (2,MASTER,5,0)

      DO 99 MAJOR=1,4

C  CLEAR THE SUBINDEX AREA.
      DO 77 I=1,10
 77   SUBIX(I)=0

C  CHANGE THE INDEX IN CURRENT USE TO SUBIX.
      CALL STINDX (2,SUBIX,10)

C  GENERATE AND WRITE NINE RECORDS.
      DO 88 MINOR=1,9
C                .
C                .
```

```
C  WRITE A RECORD.
 88    CALL WRITMS (2,RECORD,50,MINOR)

C  CHANGE BACK TO THE MASTER INDEX.
       CALL STINDX (2,MASTER,5)

C  WRITE THE SUBINDEX TO THE FILE.
       CALL WRITMS (2,SUBIX,10,MAJOR,0,1)

 99    CONTINUE

C  READ THE 5TH RECORD INDEXED UNDER THE 2ND SUBINDEX.
       CALL READMS (2,SUBIX,10,2)
       CALL STINDX (2,SUBIX,10)
       CALL READMS (2,RECORD,50,5)
C               .
C               .
C  (MANIPULATE THE SELECTED RECORD AS DESIRED.)
C               .
C               .

       STOP
       END


       PROGRAM MS5 (INPUT,OUTPUT,TAPE9)

C  CREATE FILE WITH NAME INDEX AND TWO LEVELS OF SUBINDEX.

       DIMENSION STATE(101), COUNTY(501), CITY(501), ZIP(100)
       INTEGER STATE, COUNTY, CITY, ZIP
 10    FORMAT (A10,I10)
 11    FORMAT (I10)
 12    FORMAT (5X,8I15)

       CALL OPENMS (9,STATE,101,1)

C  READ MASTER DECK CONTAINING STATES, COUNTIES, CITIES

C  AND ZIP CODES.
       DO 99 NRSTATE=1,50
       READ 10,STATNAM, NRCNTYS

C  CLEAR THE COUNTY SUBINDEX.
       DO 21 I=1,501
 21    COUNTY(I)=0
```

```
      DO 98 NRCN=1,NRCNTYS
      READ 10, CNTYNAM, NRCITYS

C  CLEAR THE CITY SUBINDEX.
      DO 31 I=1,501
 31   CITY(I)=0

      CALL STINDX (9,CITY,501)

      DO 97 NRCY=1,NRCITYS
      READ 10, CITYNAM, NRZIP

      DO 96 NRZ=1,NRZIP
 96   READ 11,ZIP(NRZ)

 97   CALL WRITMS (9,ZIP,100,CITYNAM)

      CALL STINDX (9,COUNTY,501)
 98   CALL WRITMS (9,CITY,501,CNTYNAM)

      CALL STINDX (9,STATE,101)
 99   CALL WRITMS (9,COUNTY,501,STATNAM)

C  FILE IS GENERATED.  NOW PRINT OUT LOCAL ZIP CODES.

      CALL STINDX (9,STATE,101)
      CALL READMS (9,COUNTY,501,≠CALIFORNIA≠)
      CALL STINDX (9,COUNTY,501)
      CALL READMS (9,CITY,501,≠SANTACLARA≠)
      CALL STINDX (9,CITY,501)
      CALL READMS (9,ZIP,100,≠SUNNYVALE≠)
      PRINT 12, ZIP

      CALL STINDX (9,STATE,101)

      STOP
      END
```

ERROR MESSAGES

Random file processing errors are fatal; the job terminates and one of the following error messages is printed:

97 INDEX NUMBER ERR

The index number key is negative, zero, or greater than the index buffer length minus one.

98 FILE ORGANIZATION OR RECORD TYPE ERR

During the initial OPENMS call, mass storage routines set the file organization as word addressable (FO=WA) and the record type to W (RT=W). A conflicting file organization or record type was specified in an external subroutine call or FILE control card.

99 WRONG INDEX TYPE

An attempt was made to open an existing file with the wrong index type parameter. File index type is permanently determined when a file is created.

100 INDEX IS FULL

WRITMS was called with a name index key, and the end of the index buffer occurred before a match was found. Either the name key is in error, or the buffer must be lengthened.

101 DEFECTIVE INDEX CONTROL WORD

This message may occur for either of two reasons:

1. An OPENMS for an existing file found the master index control word has been destroyed. Since this word was properly set when the file was last closed, the user should check for an external cause of file destruction.
2. A READMS or WRITMS call has encountered a defective index control word. Check for an improperly cleared sub-index array, for a program sequence that writes into an index array (other than the required initial zeroing) or for an external cause of file destruction.

102 RECORD LENGTH EXCEEDS SPACE AVAILABLE

1. During an OPENMS call, not enough index buffer space was provided for the master index of an existing file.
2. During a WRITMS call with in-place rewrite requested (r= +1), the new record length exceeded the old record length.

103 6RM/7DM I/O ERR NUMBER 000

Record Manager has detected an error; the actual error number appears in the message. Refer to Record Manager Reference Manual to identify the source of the error.

104 INDEX KEY UNKNOWN

No data record exists for the user's index key. This error may be diagnosed for a READMS call or for a WRITMS call with rewrite requested (r= +1).

COMPATIBILITY WITH PREVIOUS MASS STORAGE ROUTINES

FORTRAN Extended mass storage routines and the files they create are not compatible with mass storage routines and files created under earlier versions of FORTRAN Extended. Major internal differences in the file structure were necessitated by adding the Record Manager interface. However, source programs are fully compatible. Any source program that compiled and executed successfully under earlier versions will do so under this version, provided that all file manipulations were and continue to be executed by mass storage routines.

RENAMING CONVENTIONS

The following information will be useful only to the assembly language programmer.

REGISTER NAMES

The compiler changes some legal FORTRAN names so that FORTRAN object code can be used as COMPASS input. When a two-character name begins with A, B, or X and the last character is 0 to 7, the compiler adds a currency symbol ($) to the name for the object code listing. (A0-A7, B0-B7, and X0-X7 represent registers to the COMPASS assembler which may be used by the FORTRAN Extended compiler).

EXTERNAL PROCEDURE NAMES (PROCESSOR SUPPLIED)

CALL-BY-VALUE

The name of a system supplied external procedure called by value is suffixed with a decimal point. The entry point is the symbolic name of the external procedure and a decimal point suffix. For example, EXP. COS. CSQRT.

The names of all external procedures called by value are listed in table 8-2 Basic External Functions, section 8, part 1. A procedure will not be called by value and the name will not be suffixed with a decimal point if it appears in an EXTERNAL statement or if the control card options T, D, or OPT=0 are specified.

CALL-BY-NAME

The call-by-name entry point is the symbolic name of the external procedure with no suffix.

The call-by-name entry point consists of the symbolic name of the external procedure suffixed with a currency symbol. For example, ABS$ MOD$ RANF$

External procedures called by name appear in section 8, part 1 under the heading Additional Utility Subprograms. Any name which appears in table 8-1 Intrinsic Functions or table 8-2 Basic External Functions will be called by name also if the control card options T, D, or OPT=0 are specified or if it appears in an EXTERNAL statement.

Different entry points are required for many FORTRAN Extended and FORTRAN RUN common library routines because of the differences in calling sequences. The compiler makes the following changes to the names of external procedures:

SYSTEMC is changed to SYSTEME
OVERLAY is changed to OVERLA4

PROGRAM AND MEMORY STRUCTURE III-9

The following table shows the general form of a FORTRAN program unit. Statements within a group may appear in any order, but groups must be ordered as shown. Comment lines can appear anywhere within the program.

PROGRAM, SUBROUTINE, or FUNCTION statement
IMPLICIT Statement
Specification Statements FORMAT Statements
Statement Functions DATA Statements NAMELIST Statements FORMAT Statements
Executable Statements FORMAT Statements NAMELIST Statements DATA Statements
End line

The following description of the arrangement of code and data within PROGRAM, SUBROUTINE and FUNCTION program units does not include the arrangement of data within common blocks because this arrangement is specified by the programmer. However, the diagram of a typical memory layout at the end of this section illustrates the position of blank common and labeled common blocks.

SUBROUTINE AND FUNCTION STRUCTURE

The code within subprograms is arranged in the following blocks (relocation bases) in the order given.

START.	Code for the primary entry and for saving A0
VARDIM.	Address substitution code and any variable dimension initialization code
ENTRY.	Either a full word of NO's or nothing
CODE.	Code generated by compiling: Executable statements Parameter lists for external procedure references within the current procedure Storage statements DO loops and optimizing temporary use
	Storage for simple variables, FORMAT statements, and program constants
	Storage for arrays other than those in common
	Storage for Hollerith constants
	One local block for each dummy argument in the same order as they appear in the subroutine statement, to hold tables used in address substitution for processing references to dummy arguments

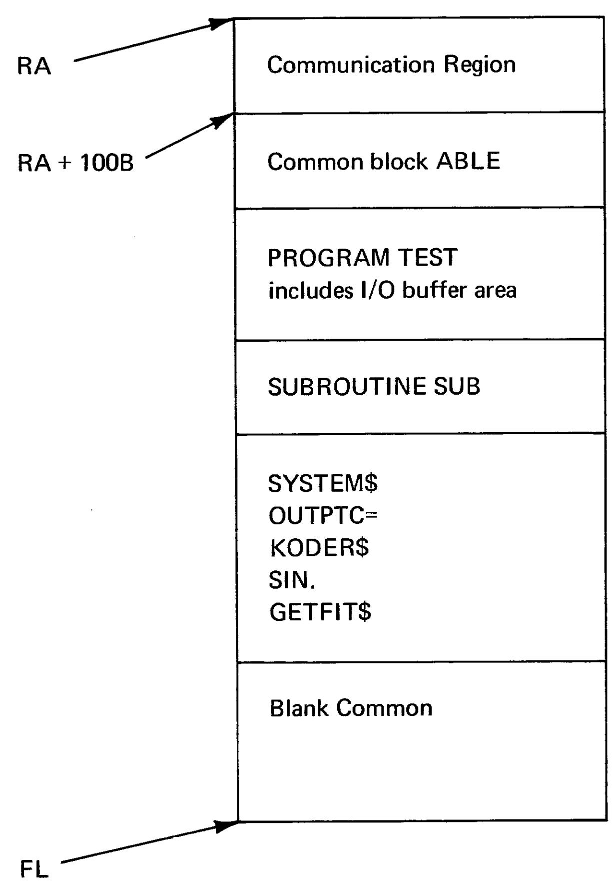
MAIN PROGRAM STRUCTURE

START.	Input/output file buffers and a table of file names specified in the program statement
CODE.	Transfer address code plus the code specified for the subroutine and function CODE. block
DATA. DATA.. HOL.	Same as SUBROUTINE and FUNCTION structure

MEMORY STRUCTURE

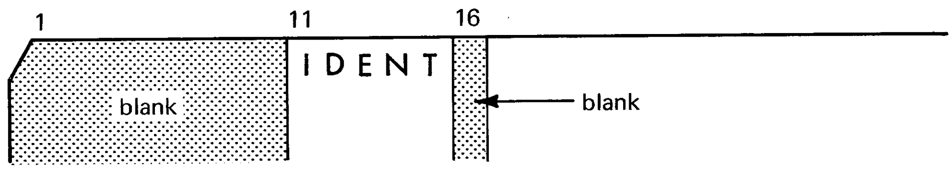
Memory is not cleared, and subprograms are loaded as they appear in the input file starting at the program's reference address (RA) + 100B, toward the user's field length (FL). RA to RA + 100B is the communication region used by the operating system. Labeled common blocks are loaded prior to the subprogram in which they are first referenced. Library routines are loaded immediately after the last subprogram and are followed by blank common.

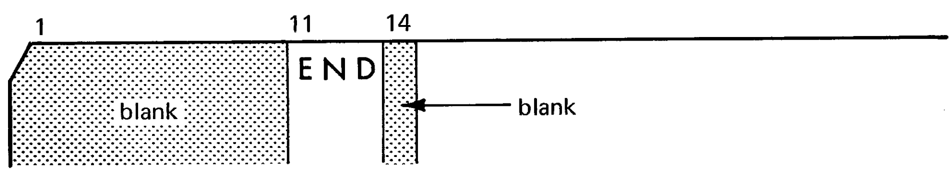
Typical memory layout:



INTERMIXED COMPASS SUBPROGRAMS

Subprograms written in COMPASS assembly language can be intermixed with FORTRAN coded subprograms in the source deck. COMPASS subprograms must begin with a card containing the word IDENTb, in columns 11-16, and terminate with a card containing the word ENDb, in columns 11-14 (b denotes a blank). Columns 1-10 of the IDENT and END cards must be blank.





ENTRY POINT

The entry point of a subprogram (for reference by an RJ instruction) is preceded by two words. The first is a trace word for the subprogram; it contains the subprogram name in left justified display code (blank filled) in the upper 42 bits and the subprogram entry address in the lower 18 bits. The second word is used to save the contents of A0 upon entry to the subprogram. The subprogram must restore A0 upon exit.

Trace word:	VFD	42/name, 18/entry address
A0 word:	DATA	0
Entry point:	DATA	0

CALL BY NAME 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	
−VFD	12/line number, 18/trace word address	
	line number	Source line number of statement containing the reference
	trace word address	Address of the trace word for the calling routine

Arguments in the call must correspond with the argument usage in the called routine, and they must reside in the same level.

The argument list consists of consecutive words of the form:

VFD 60/address of argument

followed by a zero word.

The sign bit will be set in the argument list for any argument entry address which is LCM or ECS.

CALL BY VALUE SEQUENCE

The compiler generates a call-by-value code sequence for references to external library functions if the function name does not appear in an EXTERNAL statement and the D, T, or OPT=0 options on the FTN control card are not specified. The name of any library function called by value or generated in line must appear in an EXTERNAL statement in the calling routine if the call-by-name calling sequence is required (section 8, part 1 lists the library functions called by value and generated in-line).

The call-by-value code sequence consists of code to load the argument addresses into X1 through X4, followed by an RJ instruction to the function. The second register loaded for a double precision or complex argument contains the least significant or imaginary part of the argument.

LIBRARY ENTRY POINT NAMES

If a library function which can be called by value is overridden by a routine coded in COMPASS, the COMPASS routine must use the library function name with a period appended as the entry point name (e.g. SIN.) to use the call-by-value calling sequence.

If the library function name which can be called by value appears in an EXTERNAL statement in the calling program, the call-by-name calling sequence is generated. For call-by-name entry points, the COMPASS routine must use the library name with S appended (e.g. SINS)

CONTROL RETURN

The COMPASS subprogram must restore the initial contents of A0 upon returning control to the calling subprogram. When the COMPASS subprogram is entered through a function reference, the result of that function must be in X6, or in X6 and X7 with the least significant or imaginary part of the double precision or complex result appearing in X7.

For a subprogram with no arguments, the calling sequence is:

```
+ RJ       Subprogram name
- VFD      12/line number,    18/trace word address
```

where the trace word is of the form

```
  VFD      42/7H routine name,    18/entry address
```

Example:

The following COMPASS subprogram which returns the field length of a FORTRAN program, can be referenced as either a subroutine or a function. In the first example, the field length is returned to the argument I in the CALL statement, and in the second example, it is returned as the value of the function IGETFL as well as the argument I.

1.

```
   PROGRAM FL(OUTPUT)
   CALL IGETFL(I)
   PRINT 1,I,I
1  FORMAT(* OCTAL FIELD LENGTH = *O5,*  DECIMAL FIELD LENGTH = *I6)
   END
```

2.

```
   PROGRAM FL(OUTPUT)
   PRINT 1,≠ FIELD LENGTH = ≠,IGETFL(I)
1  FORMAT (2A10,I6)
   END
```

COMPASS SUBPROGRAM

```
         IDENT   IGETFL
         ENTRY   IGETFL        Location 76 contains the field length
RA.FL    EQU     76B           of the FORTRAN program
IGETFL   BSSZ    1
         SA2     RA.FL         Field length is moved to X6 (function
         BX6     X2            results must be in X6)
         SA6     X1
         EQ      IGETFL        The field length is stored at the address
         END                   of the subroutine argument contained
                               in X1.
```

Output:

```
OCTAL FIELD LENGTH = 70000  DECIMAL FIELD LENGTH =  28672
```

FORTRAN — INTERCOM INTERFACE III-11

When a program is entered at an INTERCOM control point, INTERCOM associates the file names specified in the PROGRAM statement with the user's remote terminal device; and all references to these files are directed to the terminal.

If the SCOPE files INPUT and OUTPUT are connected, FORTRAN READ, WRITE, and PRINT statements can be used for terminal communication. The user can specify other files to be associated with the terminal with calls to CONNEC and DISCON.

INTERCOM terminal input/output is supported for formatted or NAMELIST input/output only.

A file can be connected to the terminal with the statement:

CALL CONNEC (lfn)

If the file is already connected, the request is ignored. If the file has been used previously, but is not connected, this request clears the file's buffer, writes an end of file, and backspaces over it before the connection is performed.

A file is disconnected by the statement:

CALL DISCON (lfn)

This request is ignored if the file is not connected. After a disconnect, the file is re-associated with its former device.

lfn internal file name:

File designator, 1 to 99

Display code file name, L format, left justified with zero fill

Simple integer variable whose value is either of the above

Examples:

```
CALL CONNEC (3LEWT)

CALL DISCON (6)

K=5LINPUT
CALL DISCON(K)

J=12
CALL CONNEC(J)
```

Any files listed in the PROGRAM statement can be connected or disconnected during program execution. An attempt to connect or disconnect an undefined file results in a fatal execution time error, and the job is terminated.

If execution diagnostics are to be sent to the terminal, the file OUTPUT must be declared in the PROGRAM statement and connected to the terminal.

Calls to CONNEC and DISCON are ignored when programs are not executed through an INTERCOM control point.

During a typical compilation and execution, the following listings are produced:

source program

reference map

core map

The compiler produces a reference map for each routine successfully compiled. In this map, the compiler generated addresses assume loading of program units starts at location 0. A description of the reference map appears in section 14, part 1.

A map is produced by the loader at load time. In this map, the user program starts at relative address 100. (The first 100 words, 0-99, serve as the communication region between the SCOPE operating system and the user program.) Refer to the Loader Reference Manual for details of the load map.

To find the address of a variable, add the address of the program unit, which appears in the load map, to the address of the variable which appears in the reference map.

All locations and addresses in the reference map and the core map are in octal.

For example,

```
VARIABLES         SN   TYPE
     0  A             REAL
    16  AVG           REAL
    17  I             INTEGER
     0  J             INTEGER
```

```
PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK            ADDRESS     LENGTH           FILE

VARDIM2              100       2200           LGO
SET                 2300         64           LGO
IOTA                2364         35           LGO
PVAL                2421         56           LGO
AVG                 2477         57           LGO
MULT                2556         40           LGO
FLOAT$              2616          3           FORTRAN-L
ABS$                2621          3           FORTRAN-L
SYSTEM$             2624        664           FORTRAN-L
```

the address of the generated code of the variable I would be:

```
 2477
  +16
 ----
 2515
```

DMPX.

When a program does not compile or execute successfully, a partial dump is produced. A DMPX includes the contents of the registers, the first 100 words of the user's field length (the communication region used by SCOPE), and the contents of the 100 (octal) words immediately preceding and immediately following the addresses where the job terminated.

1.	P	Address of program step to be executed next if job had not terminated
2.	RA	Reference address: absolute address where user's field begins. All other addresses are relative to this address.
3.	FL	Field length of job
4.	EM	Default exit mode
5.	RE	Extended core storage reference address
6.	FE	Field length assigned to job in extended core storage
7.	MA	Address in SCOPE monitor routine used for linkage between SCOPE and user program
8.		Address registers
9.		Contents of address registers
10.		Index registers
11.		Contents of index registers
12.		Operand registers. X1-X5 contain operands used in calculations; registers X6 and X7 contain the results of calculations.
13.		Contents of operand registers
14.		Contents of locations specified in the A register. For example, items 8 and 9 show register A2 contains the address 002155, and item 14 shows the address A2 contains 1725 2420 2524 0000 0133.
15.		Address of 60-bit word in central memory, followed by contents of that word (in octal)
16.		Indicates that contents of previous locations are repeated up to but not including this location

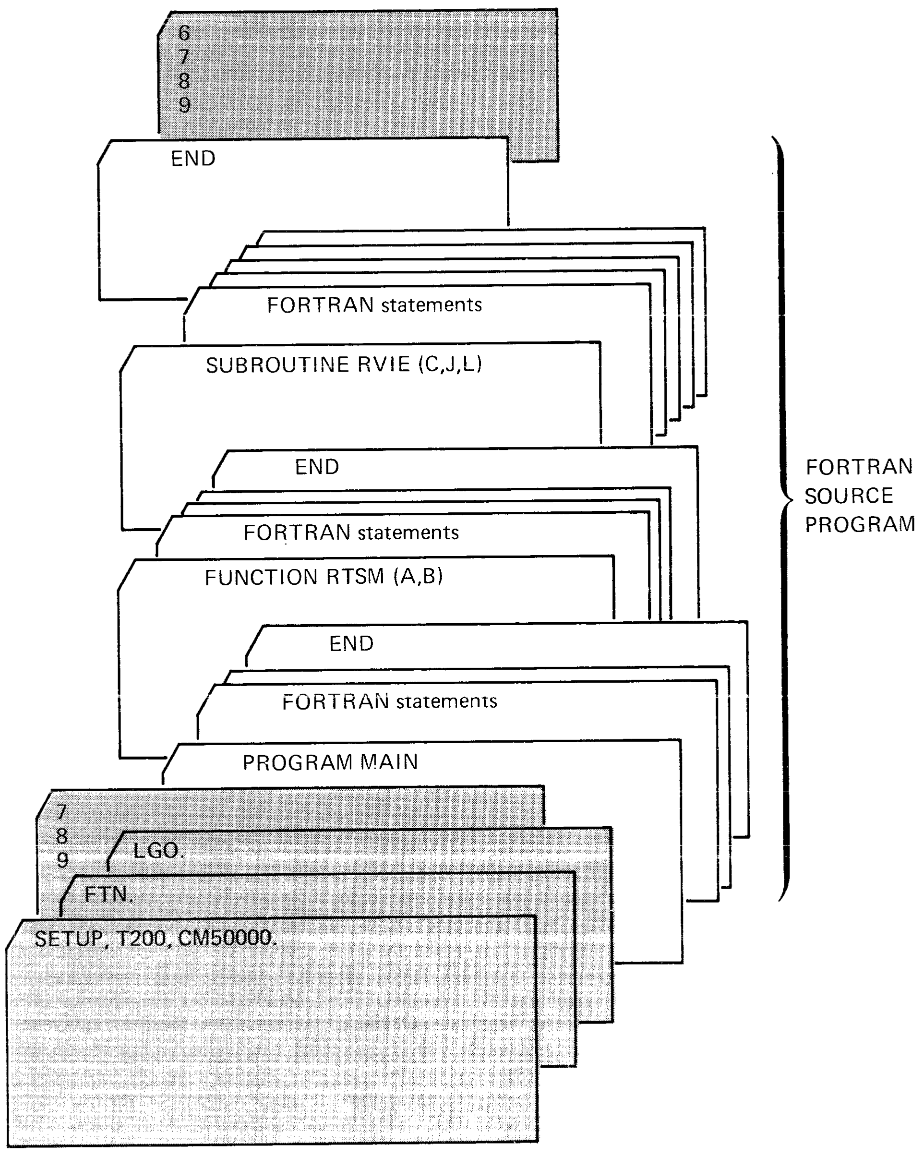
```
                DMPX.        (8)      (9)      (10)     (11)                                                       (14)
(1) P     013552   A0   002133   B0   000000
(2) RA    312100   A1   000001   B1   000001       C(A1)=  0000  0000  0000  0000  0000      C(B1)=  0000  0000  0000  0000  0000
(3) FL    065000   A2   002155   B2   000001       C(A2)=  1725  2420  2524  0000  0133      C(B2)=  0000  0000  0000  0000  0000
(4) EM    070000   A3   002140   B3   000040       C(A3)=  0000  0000  0100  0000  2165      C(B3)=  0000  0000  0000  0000  0000
(5) RE    000000   A4   004474   B4   000130       C(A4)=  0000  0000  0000  0000  0004      C(B4)=  0000  0000  0000  0000  0000
(6) FE    000000   A5   002135   B5   000001       C(A5)=  0000  0000  0240  4000  0000      C(B5)=  0000  0000  0000  0000  0000
    MA    001400   A6   000001   B6   004636       C(A6)=  0000  0000  0000  0000  0000      C(B6)=  5140  0044  7404  0000  4613
(7) (12)           A7   002140   B7   002206       C(A7)=  0000  0000  0100  0000  2165      C(B7)=  0000  0000  0000  0000  0000
    X0    7777   7777   7777   7777   7776
    X1    0000   0000   0000   0000   0000
    X2    0000   0000   0000   0000   4776
    X3    0000   0216   5000   0000   0004
    X4    0000   0000   0000   0000   0005
    X5    0000   0000   0000   0000   0002
    X6    0102   2400   0000   0000   0000
    X7    0000   0000   0100   0000   2165
                        (13)

(15) 00000    00000 00000 00000 00000    00002↱11162 02524 00000 00100          17252 42025 24000 02133
     00004    00000 00000 00000 00000          32323 23232 22140 00322          03171 52023 00000 00305          32323 23232 22150 00337
     00010    32323 23232 17200 00354    (16)  00000 00000 00000 00000          00000 00000 00000 50167          00000 00000 00000 50121
     00014    00000 00000 00000 00000    00017↱23312 31726 14000 00001
     00020    01022 14000 00000 05152          03141 72305 35530 00000          00000 00000 00000 00000
     00024   ↱00000 00000 00000 00015          00000 00000 00000 00000    00032↱00000 00000 00000 50064          00000 00000 00000 00000
     00037   ↱00000 00000 00000 00007
     00040    00000 00000 00000 00000    00043↱00000 00000 00000 50106
     00044    00000 00000 00000 00000    00046↱55555 55555 55555 55542          00000 00000 00000 00000    00052↱00000 00000 00000 00002
     00053    00000 00000 00000 00000
     00054    51100 00001 03110 00054          64550 02500 00000 46000          00000 00000 00000 00000          13443 03040 00060 67402
     00060    56124 63310 13415 21422          61447 77776 03040 0(056          07040 00060 51600 00001          00000 00000 00000 00021
     00064    14071 70000 00000 00000          00000 00000 00000 13607          40000 00000 00000 00100          00000 00000 40000 00000
     00070    00000 00000 00000 00000
     00100   ↱11162 02524 00000 00000

     13452    50100 00021 20101 46000          03210 13456 50100 00002
     13454    43744 20130 15117 37341          03330 13456 10411 46000          50100 00015 20101 46000          03210 13476 37224 46000
     13460    03220 13465 66340 46000          50100 00020 43770 20106          11771 20766 54710 14422          61600 13456 50100 00021
     13464    63510 02500 11034 46000          27704 51300 07654 40773          26707 36377 63373 37443          20302 37443 03040 10710
     13470    50100 00020 43770 20106          15117 36441 72147 77765          03310 13473 10411 66331          50100 00020 43770 20106
     13474    11771 12774 03345 13474          20766 54710 04000 10710          50500 00015 20530 73550          50100 00017 43770 20106
     13500    15117 63510 50100 00006          43744 20130 15117 46000          50300 00015 73330 37114          53550 37113 03210 13507
     13504    36441 71600 00142 46000          50100 00015 43701 12771          54710 46000 61000 46000          03040 10710 36334 46000
     13510    50100 00015 43752 11771          12773 03335 13511 54710          56330 50100 00020 43770          20106 15117 63310 46000
     13514    63440 61600 13530 37124          03210 12246 37442 46000          50100 00020 43752 20130          11771 12774 03345 13517
     13520    20744 54710 63420 46000          61600 13522 04000 12246          64330 27444 61600 13525          50100 00021 63510 10577
     13524    02500 11034 61000 46000          26454 56330 63340 46000          50400 00020 20430 73440          04000 13514 61000 46000
     13530    64330 50100 00020 43770          20106 11771 12774 46000          03345 13532 20766 54710          50100 00015 43752 20130
     13534    11771 74150 12771 46000          03315 13535 20744 54710          50100 00017 43770 20106          11771 76140 12771 46000
     13540    03315 13540 20766 54710          50100 00015 03210 10710          54301 43744 20330 15337          50100 00015 73110 37431
     13544    50100 00015 76710 20772          12771 54710 50100 00015          76710 20770 15717 54710          61600 13456 04000 11052
     13550    04000 13563 00000 00000          01300 00000 00000 00000          04000 04474 00000 00000          51100 00001 03110 13553
     13554    54610 04000 13551 46000          51100 00066 03310 13557          51100 13550 04000 13560          71100 00130 20160 46000
     13560    13661 13161 13661 46000          51600 13551 10611 46000          51100 00001 01000 13550          20652 01000 13552 46000
     13564    51100 00001 03110 13564          04004 13565 61000 46000          51100 00001 03110 13565          71602 20314 04000 13563
     13570    20150 36661 01000 13552          04000 07354 00000 00000          71602 20314 20652 36662          53160 20173 03310 13571
     13574    03010 13571 51100 00001          03110 13573 71100 00001          04000 13570 61000 46000          71603 24616 12661 20651
     13600    01000 13552 61000 46000          04004 13601 61000 46000          20622 12161 73610 20123          03210 13577 20151 13116
     13604    20636 51600 13606 74660          36116 20123 04000 13577          00000 00000 00000 00000          71603 24616 12661 20651
     13610    00000 00000 00000 00000    13652↱00000 00000 00000 00000
```

```
XJP DUMP
P   00 000227 A0 061000 BP 000000  SC(A0)=                          SC(P )= 0130 0000 0700 0000 0320
RAS 00 015400 A1 000004 B1 000001  SC(A1)= 2000 0000 0000 0000 0017 SC(B1)= 0000 0000 0000 0000 0000
FLS 00 061000 A2 000005 B2 000000  SC(A2)= 0000 0000 0000 0000 0000 SC(B2)= 0000 0000 0000 0000 0000
PSD 00 060000 A3 000000 B3 000000  SC(A3)= 0000 0000 0000 0000 0000 SC(B3)= 0000 0000 0000 0000 0000
RAL 00 257000 A4 000000 B4 000001  SC(A4)= 0000 0000 0000 0000 0000 SC(B4)= 0000 0000 0000 0000 0000
FLL 00 020000 A5 000126 B5 000111  SC(A5)= 0006 0452 0000 0000 0325 SC(B5)= 0000 0000 0000 0000 0312
NEA 00 015020 A6 000120 B6 000000  SC(A6)= 0211 1600 0000 0000 0000 SC(B6)= 0000 0000 0000 0000 0000
EEA 00 010460 A7 000000 B7 000000  SC(A7)= 0000 0000 0000 0000 0000 SC(B7)= 0000 0000 0000 0000 0000

X0  7700 0000 0000 0000 0000       SC(X0)= 0000 0000 0000 0000 0000 LC(X0)= 0000 0000 0000 0000 0000
X1  0000 0000 0000 0011 6174       SC(X1)=                          LC(X1)=
X2  0000 0000 0000 0000 0000       SC(X2)= 0000 0000 0000 0000 0000 LC(X2)= 0000 0000 0000 0000 0000
X3  7777 7777 7777 7777 7775       SC(X3)=                          LC(X3)=
X4  7777 7777 7777 7777 7774       SC(X4)=                          LC(X4)=
X5  0000 0000 0000 0000 0000       SC(X5)= 0000 0000 0000 0000 0000 LC(X5)= 0000 0000 0000 0000 0000
X6  0211 1600 0000 0000 0000       SC(X6)= 0000 0000 0000 0000 0000 LC(X6)= 0000 0000 0000 0000 0000
X7  0000 0000 0000 0000 0000       SC(X7)= 0000 0000 0000 0000 0000 LC(X7)= 0000 0000 0000 0000 0000


SC 000000  00000000000000000000 00000000000000000000 24051520000000000001 02111600000000000001               TEMP     ABIN      A
SC 000004  20000000000000000017 00000000000000000000 23240000000000000002 03032000000000000017  P      O      ST       BCCP      O
SC 000010  00000000000000000000 00000000000000000010 00000000000000726630 00000000000000003600               H         X        3
SC 000014  00000000000000000000 00000000000000000000 00000000000000013624 00020330000000013624                       A3T BCX   A3T
SC 000020  00000000000000003034 00621000000000634001 00000000000000000001 40000000400000005236         X1  H    5A     A5   5    )3
SC 000024  00000000000000000000 00000000000000000000 00000000000000004452 00621000000000634001                       9)  H     5A
SC 000030  00000000000000000001 00000000000000000000 00000000000000000000 00000000000000000000         A
SC 000034  00000000000000000000 03172031232700000000 00000000000000000000 00000000000000000000          COPYSW
SC 000040  00000000000000000000 03172031232700000000 00000000000000000000 00000000000000000000          COPYSW
SC 000044  00000000000000000000           EQUAL THRU 000063
SC 000064  03172031232700000003 00000000000000000000 00000000000000000000 03032000000000000002  COPYSW   C                 CCP       B
SC 000070  03172031232751240515 20550211165620520010 00000000000000000000 00000000000000000000  COPYSW(TEMP,BIN,P)
SC 000074  00000000000000000000 00000000000000000010 00000000000000000000 53000000000077000165                              $         A
SC 000100  24051520000000000000 00007646004201000644 00000000554020000000 00000000000000000232  TEMP        - 7A F9    5P             BZ
SC 000104  00000000000000000000 00000000004000000000 00060452000000000325 00000000000000000000                5      FD)   CU
SC 000110  00000000000000000000 00000000000000000312 00000000000000000112 00000000000000000254                   CJ          AJ       B=
SC 000114  00001000000000000000 00000000000000201054 00000000000000000000 00000000000000000000    H             PH
SC 000120  02111600000000000000 00007646000002001000 00000000554000000000 00000000000000000000  BIN         - B H       5
SC 000124  00000000000000000000 00000000004000000000 00060452000000000325 00000000000000000000                5      FD)   CU
SC 000130  00000000000000000000 00000000000000000315 00000000000000000222 00000000000000000000                   CM          BR
SC 000134  00001000000000000000 00000000000000176001 00000000000000000000 00000000000000000000    H             0 A
```

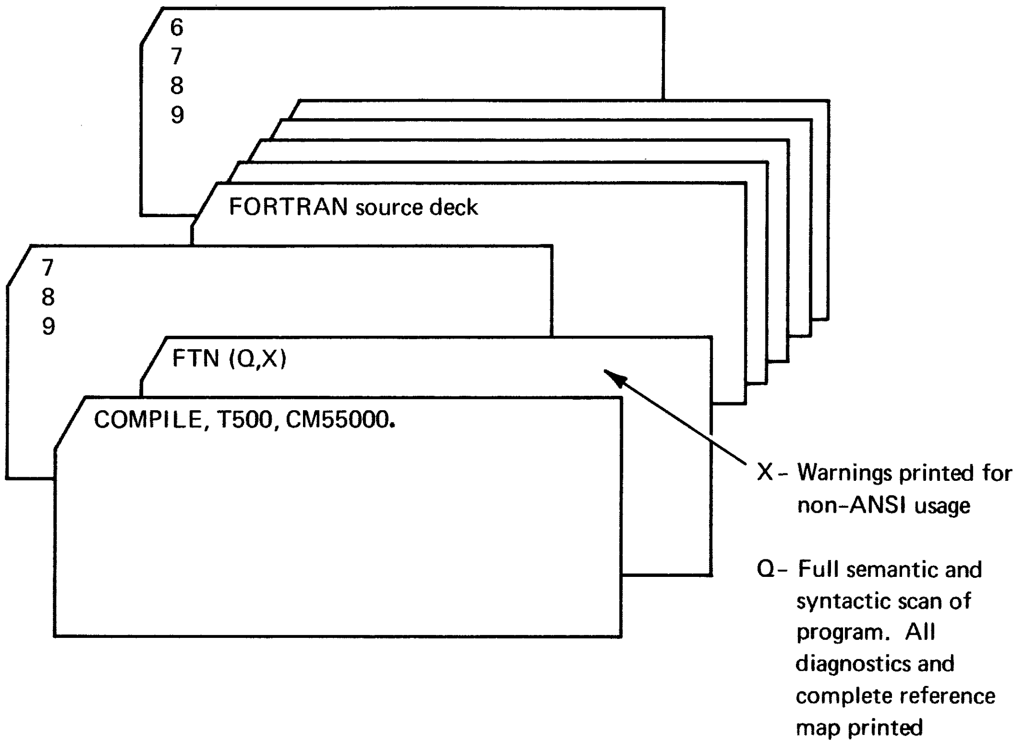
```
                                        LOADER   VER.  1.0                                                            PAGE   1
PROGRAM  WILL BE ENTERED AT  VARDIM2     (    137)                    SCM LENGTH      4405          LCM LENGTH          0

         BLOCK          ADDRESS     LENGTH

         VARDIM2          100         150
         SET              250          60
         PVAL             330          35
         IOTA             365          37
         AVG              424          20
         MULT             444          21
         GETFIT$          465          31
         KODER$           516        1406
         NAMOUT=         2124         555
         OUTPTC=         2701         247
          /SCOPE2/       3150           0
         SYSTEM$         3150        1071
          //             4241         144
          //             4241         144

         ENTRY          ADDRESS   REFERENCES

         VARDIM2
           TAPE6=           100
           OUTPUT=          100
           VARDIM2          137
         SET
           SET              252   VARDIM2       141
           INC              276   VARDIM2       145
         PVAL
           PVAL             332   VARDIM2       147        151
         IOTA
           IOTA             367   VARDIM2       143
         AVG
           AVG              426   VARDIM2       175
                                  MULT          454
         MULT
           MULT             446   VARDIM2       200
         GETFIT$
           GETFIT$          467   NAMOUT=      2133
                                  OUTPTC=      2722
         KODER$
           KODER.           517   OUTPTC=      2715       2771       3007       3054
         NAMOUT=
           NAMOUT.         2130   VARDIM2       153
           OUTPTN          2644
         OUTPTC=
           OUTCI.          2720   VARDIM2       155
           OUTCR.          3016
           OUTCX.          3022
           OUTPTC          3030
         SYSTEM$
           LIN.LIM         3154   NAMOUT=      2351
                                  OUTPTC=      3116
           MSG84           3155   NAMOUT=      2542
                                  OUTPTC=      3073
```

FORTRAN SOURCE PROGRAM WITH SCOPE CONTROL CARDS

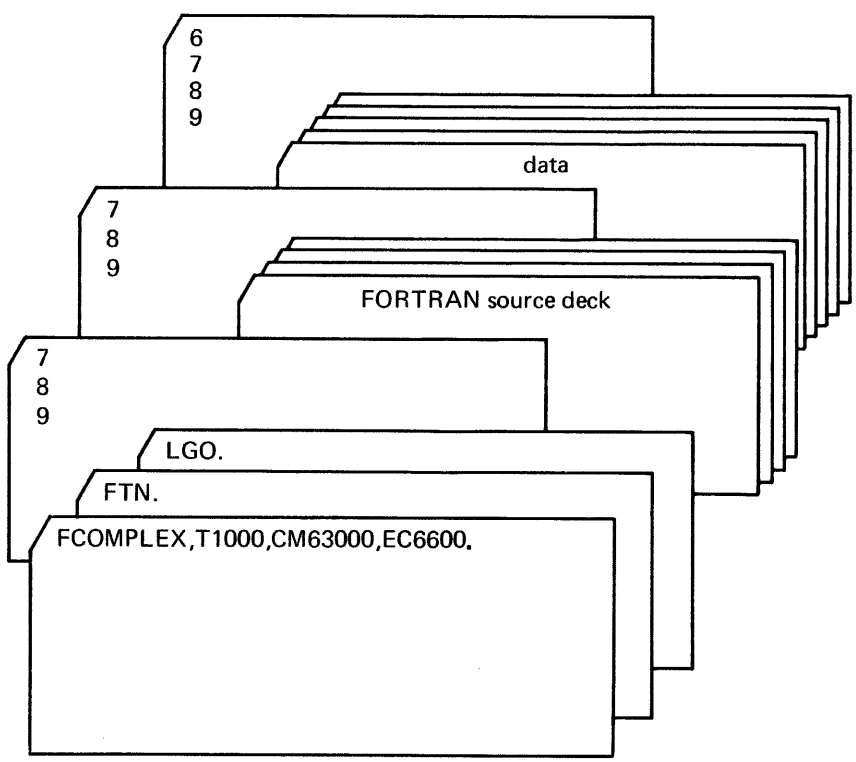
In the following sample deck SCOPE control cards are shaded. Refer to the SCOPE Reference Manual for details of these cards.



COMPILATION ONLY

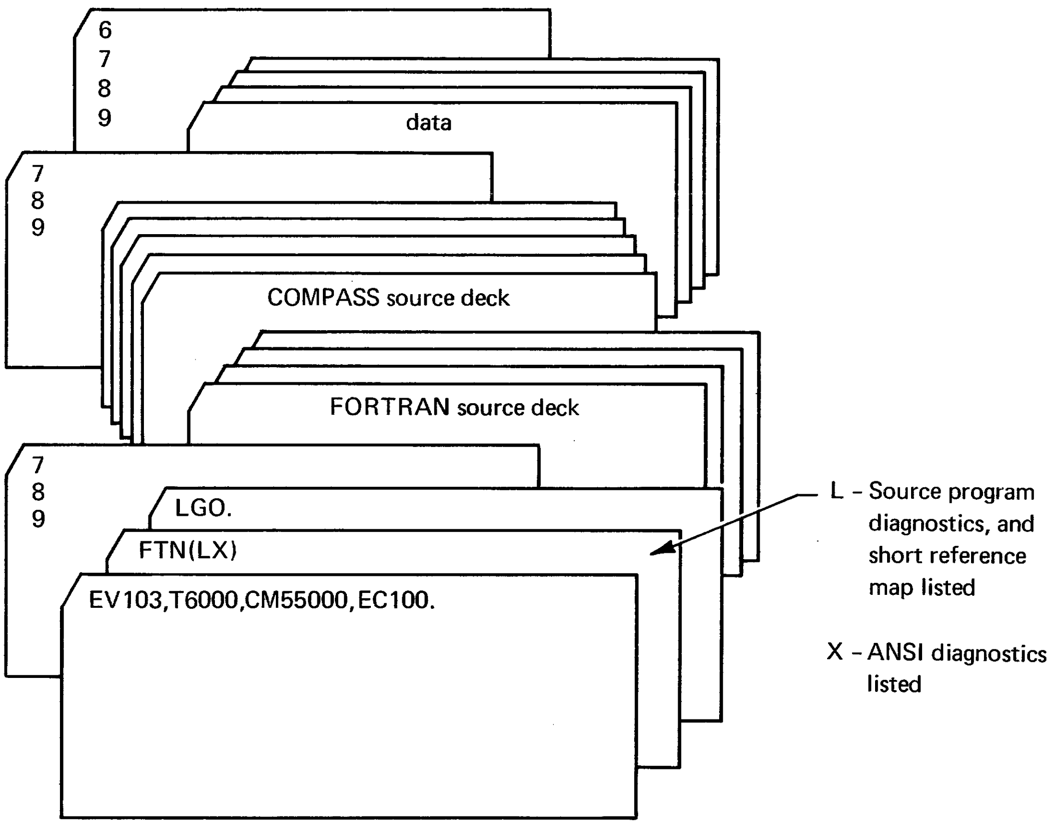


COMPILATION AND EXECUTION

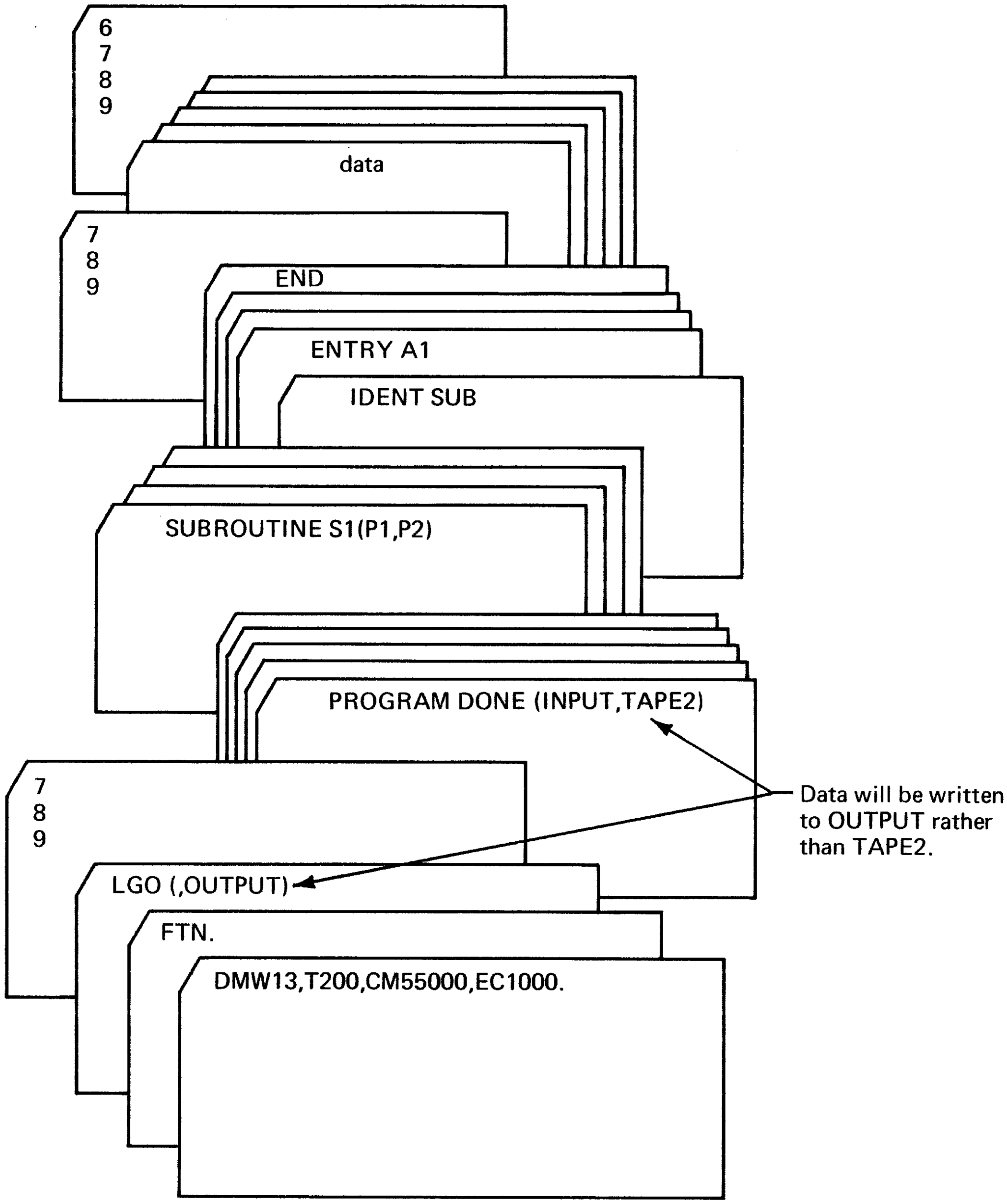


FORTRAN COMPILATION WITH COMPASS ASSEMBLY AND EXECUTION

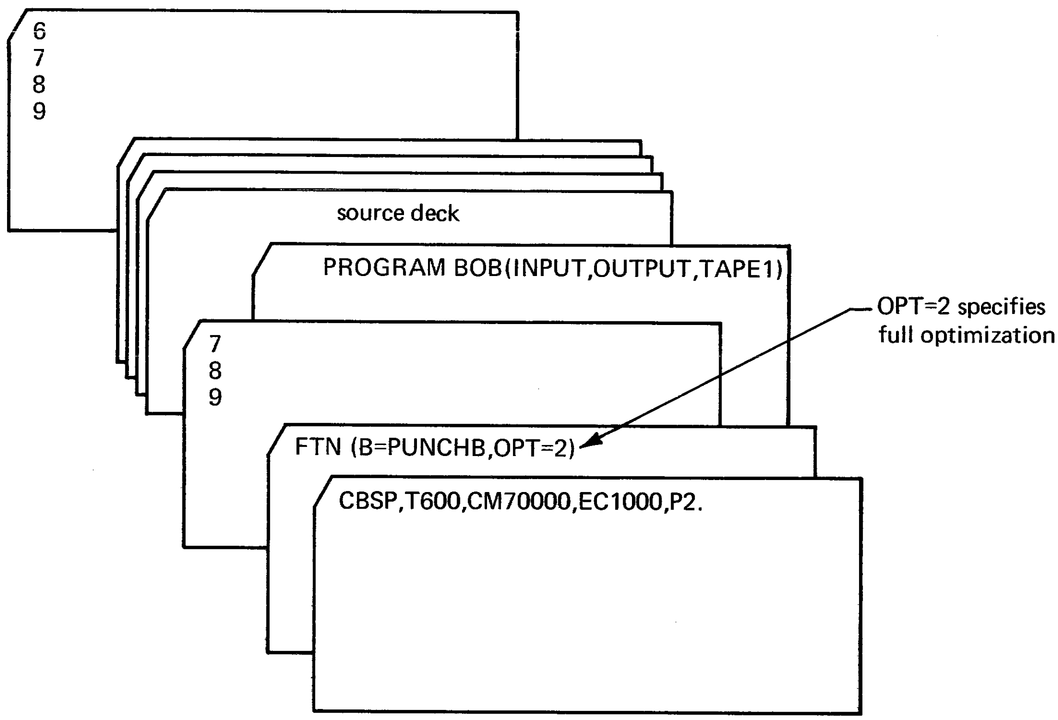
FORTRAN and COMPASS program unit source decks can be in any order. COMPASS source decks must begin with a card containing the word IDENTb in columns 11-16 and terminate with a card containing the word ENDb in columns 11-14 (b denotes a blank). Columns 1-10 of the IDENT and END cards must be blank.



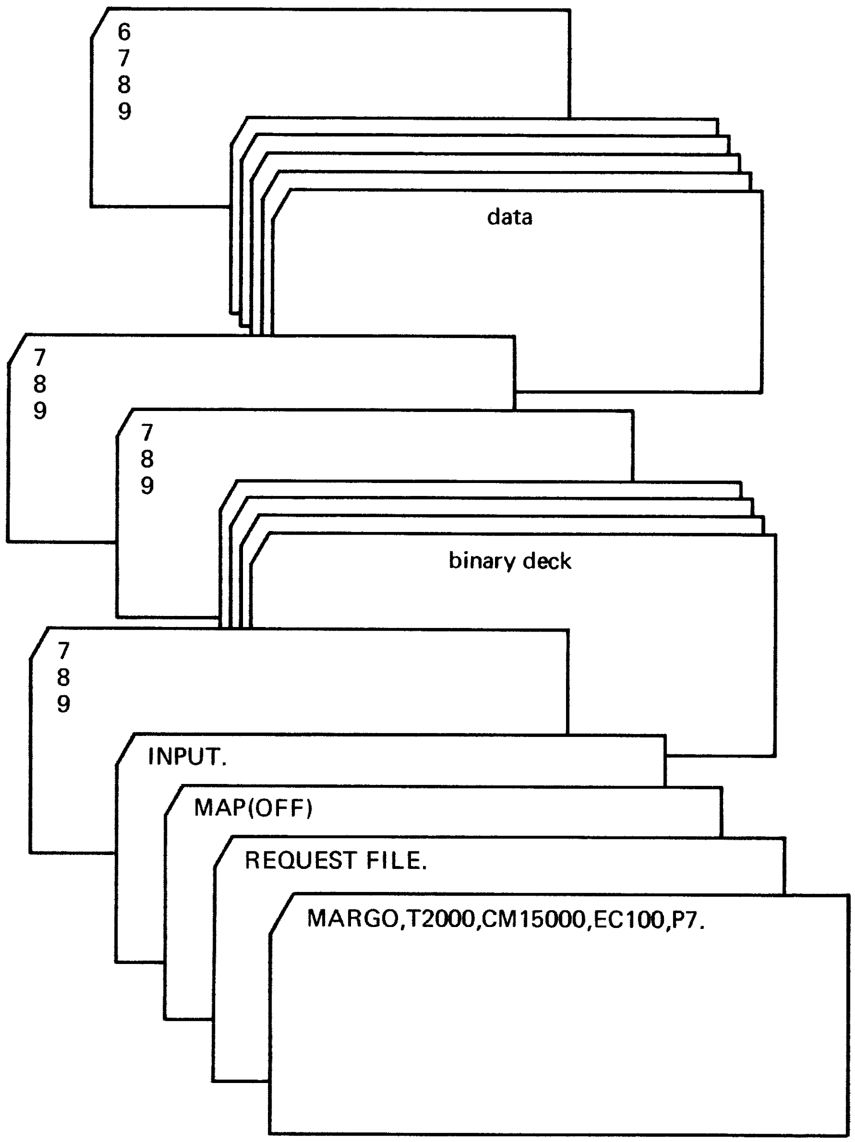
COMPILE AND EXECUTE WITH FORTRAN SUBROUTINE AND COMPASS SUBPROGRAM



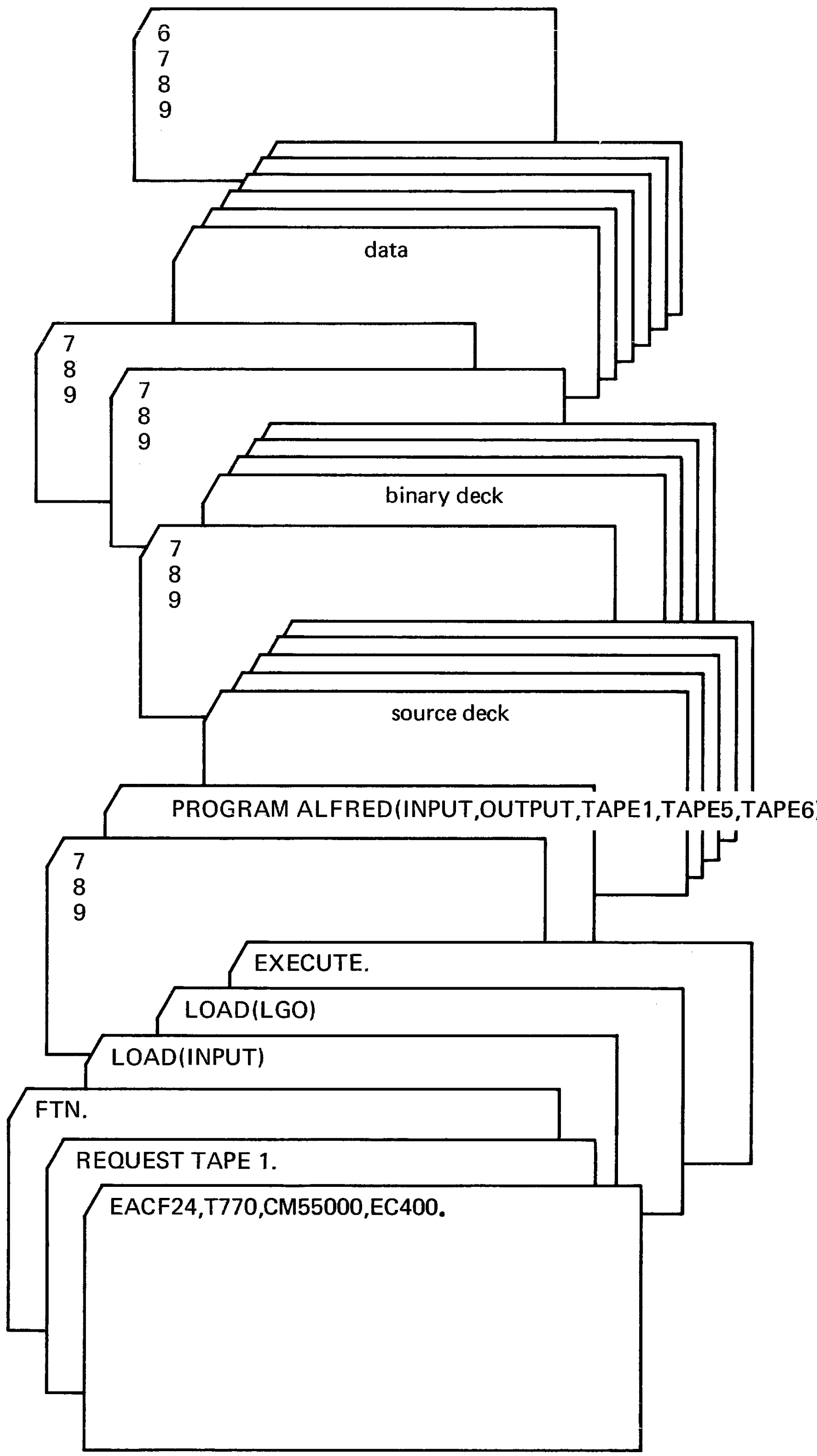
COMPILE AND PRODUCE BINARY CARDS



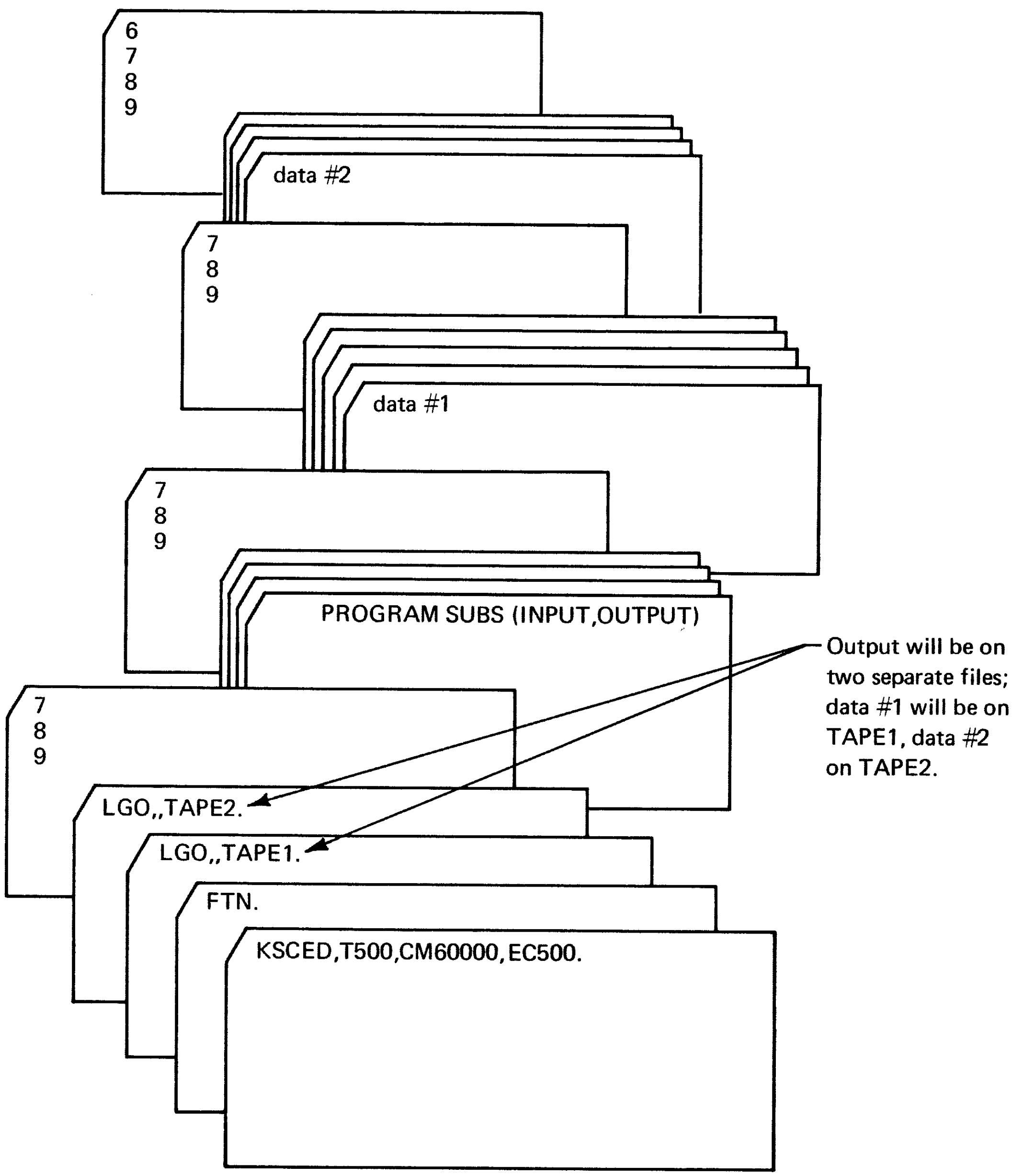
LOAD AND EXECUTE BINARY PROGRAM



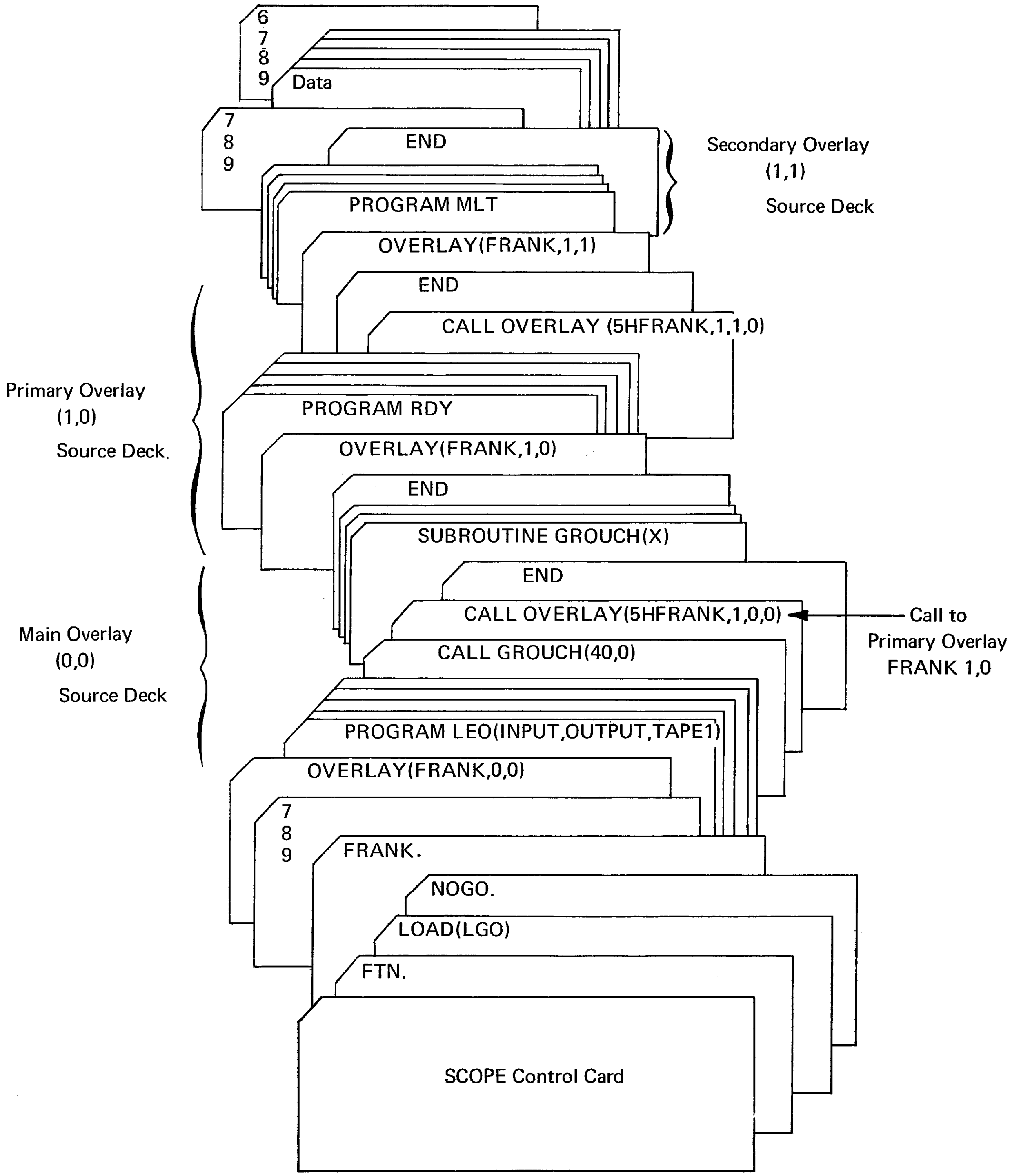
COMPILE AND EXECUTE WITH RELOCATABLE BINARY DECK



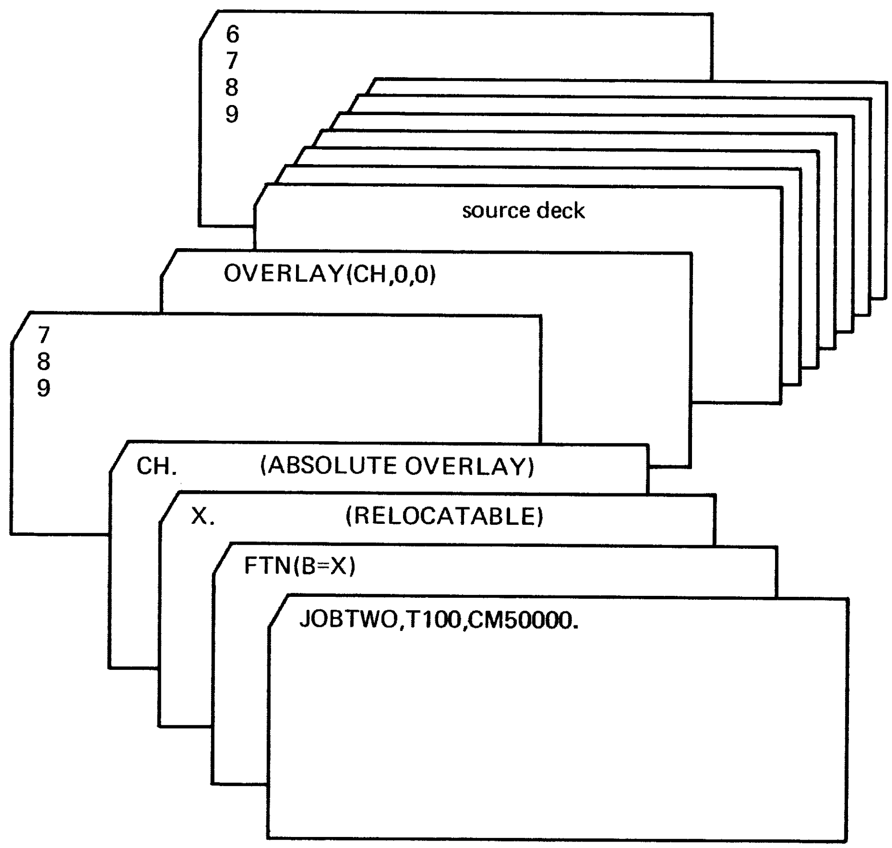
COMPILE ONCE AND EXECUTE WITH DIFFERENT DATA DECKS



PREPARATION OF OVERLAYS



COMPILATION AND 2 EXECUTIONS WITH OVERLAYS



INDEX

CUT ON THIS LINE

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: FORTRAN Extended 6000 Version 4/7000 Version 2
Reference Manual

PUBLICATION NO. 60305600 REVISION A

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
Software Documentation
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD FOLD

STAPLE STAPLE

CONTROL DATA

►►CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

1/2 3/4 1 1-1/4

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. 60305600

Litho in U.S.A.