HETEROGENEOUS ELEMENT PROCESSOR

FORTRAN

EXTERNAL DESIGN REPORT

SEPTEMBER 7, 1977
(REVISED OCT.11, 1977)
(REVISED NOV.21, 1977)

RYAN-McFARLAND CORPORATION
603 SILVER SPUR ROAD
ROLLING HILLS ESTATES,
CALIFORNIA 90274

PAGE NUMBER

INTRODUCTION

INTRODUCTION

This is the external design report for the Heterogeneous Element Processor (HEP), implemented by the Ryan-McFarland Corporation under Denelcor purchase order No. 014882. It is directed to the FORTRAN programmer as well as the programming staff responsible for maintaining the compiler. Its purpose is to explain the HEP version of the FORTRAN language, the various inputs and outputs of the compiler, and to give some examples as well.

1. FORTRAN SOURCE PROGRAM FORM

A FORTRAN source program consists of one main program and any number of subprograms. The main program and subprograms are made up of statements using the FORTRAN character set.

1.1 Character Set

The character set has two subsets: alphanumeric characters and special characters.

alphanumeric characters:
Letters (A-Z) and
Digits (0-9)

special characters:

	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Currency symbol
'	Apostrophe

Blanks may appear anywhere in a source program. They are significant only in Hollerith constants and format specifications.

1.2 Source Statements

Standard FORTRAN statements are accepted as formatted 80-character or less lines (or records). Each line is divided into four fields:

Columns	Field
1-5	Statement Label
6	Continuation Indicator
7-72	Statement
73-80	Identification

The identification field is ignored by the FORTRAN compiler and is provided for the convenience of the programmer.

1.2.1 Statement Labels

The statement label is made up of digits placed anywhere in columns 1-5 of the initial line of a statement. Blanks and leading zeros are ignored.

1.2.2 Statements

A statement consists of an initial line and any number of continuation lines.

An initial line is a line that is neither a comment line nor an END line and contains either a blank or the digit 0 in Column 6. A continuation line is not a comment line and contains any character other than blank or 0 (zero) in Column 6. An END line is a line containing an END statement, which cannot be continued.

EXAMPLE:

1	2	3	4	5	6	7	8	9	10	11	12 . . .
		2	0	0		A	=	B	+		
					X	C					
					X	+	D				

The first line of this example is an initial line. The second and third lines are continuation lines. The statement label is 200. The statement is equivalent to:

A = B + C + D

1.2.3 Comments

Comment lines have the character C or an asterisk (*) in Column 1. Comments are for the convenience of the programmer and permit him to describe the program; they do not influence the program. Columns 2-72 may be used in any desired format.

A comment line can only be followed by another comment line or the initial line of a statement.

1.2.4 Blank Lines

Blank lines may be included in the source text. Columns 1-72 must be blank. A blank line can only be followed by a comment line, another blank line or the initial line of a statement.

1.2.5 Source Statement Ordering

Table 1.1 shows the order in which source statements of each program must be written. Within each group, the statements may be written in any sequence.

DATA and CONST statements may appear anywhere after Group 2 and before Group 6, but must appear after any declarations (COMMON, DIMENSION, or type) affecting the variables to be initialized.

FORMAT statements may appear anywhere before Group 6.

1.3 D in Column 1, Conditional Compilation

If the compiler option for conditional compilation is specified, all source lines with a D in Column 1 are compiled as though Column 1 contained a blank. Otherwise these lines are treated as comments.

SOURCE STATEMENT	GROUP
PROGRAM FUNCTION SUBROUTINE BLOCK DATA	1
IMPLICIT	2
COMMON COMPLEX DIM[ENSION] DOUBLE [PRECISION] EQUIV[ALENCE] EXTERNAL INTEGER LOGICAL MAX REGS PRIVATE REAL REG[ISTER]COM[MON]	3
Statement Function	4
Assignment ASSIGN Assign and Unconditional GO TO BACKSPACE CALL Computed GO TO CONTINUE CREATE DECODE DO ENCODE ENDFILE IF PAUSE PURGE READ RESUME RETURN REWIND STOP WRITE	5
END	6

Table 1.1

1.4 COPY Statement

The copy feature enables parts of the program to be stored in more than one file. This statement appears as follows:

COPY name

where "name" is a file pathname. The name must be completely contained on one line (not extended across continuation lines) and may not contain blanks.

A program may contain any number of COPY statements, but they must not be nested deeper than three.

The contents of the named file are inserted into the source program such that the first record of the file will be the next line after the COPY statement. Thus COPY statements may be labelled and referenced the same as CONTINUE statements. A copy statement may not precede group 1 statements, nor may it follow a group 6 statement.

2. DATA

Data is represented as constants and variables. A constant is a quantity whose value is explicitly stated. A variable is a quantity whose value may vary and is referenced by an identifier which symbolically identifies the variable.

2.1 Identifiers

Identifiers are used to give names to:

-- Variables
-- Subprograms
-- Common blocks

An identifier is a string of alphanumeric characters, the first of which must be alphabetic. Any number of characters are allowed in an identifier but only the first eight are recognized. Certain identifiers may be preceded by a dollar sign ($) (see section 2.2). The dollar sign is not included as one of the significant characters of an identifier.

EXAMPLES:

```
X15
PERMUTATION
STRAIN
```

2.2 Variables

Variable names are identifiers that represent quantities which may assume a number of different values. Each variable has an associated data type: integer, logical, real, double precision, complex. The variable may only assume values of that type.

A variable may be either synchronous (the FORTRAN standard) or asynchronous. An asynchronous variable is actually a pair: A standard variable and a semaphore. Assignment to the variable causes a wait until it is semaphored as empty and leaves it semaphored as full with the new value. Conversely, each use causes a wait until the variable is full, and leaves it empty. The semaphore may be interrogated with an intrinsic function (FULL or EMPTY) to determine its state. An asynchronous variable is indicated by a dollar sign ($) preceding the first character of the identifier. Also, the same variable may not be used as both a synchronous and an asynchronous variable; i.e., $A and A may not appear in the same program unit.

Variables may be simple or arrays.

2.2.1 Simple Variables

A simple variable name identifies the location in which a variable value can be stored.

EXAMPLES:

```
N
X4
$VALUE
LOOK UP
```

2.2.2 Arrays

An array variable name identifies an ordered set of data having one, two, or three dimensions. Each element of an array is referenced by the array name followed by a set of subscripts.

An array has an associated type the same as a simple variable. An array must be declared by an array declarator which establishes the number of dimensions and the size of each. Typing of an array is done the same as for a simple variable.

The declaration of an array's type may occur separately from the declaration of its dimensions.

2.2.2.1 Array Storage Allocation

The multi-dimensional arrays declared by the programmer are assigned to the one-dimensional computer memory in such a way that the left-most subscript varies most rapidly and the right-most the least.

EXAMPLE:
Allocation of array K of three dimensions of two elements each.

Memory Sequence	Array Element
1	K(1,1,1)
2	K(2,1,1)
3	K(1,2,1)
4	K(2,2,1)
5	K(1,1,2)
6	K(2,1,2)
7	K(1,2,2)
8	K(2,2,2)

2.2.2.2. Array References

Most FORTRAN statements operate upon only one element of an array at a time. A member of an array is referenced in the form:

$$\text{array}\ (S_1, S_2, S_3)$$

where "array" is the name of the array, and the S_i are subscript expressions. A subscript expression may be any expression (see section 4.1). The expression is converted to type integer after evaluation.

EXAMPLES:

```
Y(1)
STATION(K)
Q(LINE(N,X)+RHO,N)
```

The value of a subscript must be within the limits specified for the array. The number of subscripts must equal the number of dimensions specified for the array.

2.3 Data Types

The following data types are defined:

Integer
Logical
Real
Double Precision
Complex

The name assigned to a variable is associated with a data type either implicitly or explicitly.

The explicit type declaration statements (section 3.3.1) assign the type explicitly.

If a variable is not explicitly typed by declaration, the compiler implicitly types it according to the following conventions:

- If the identifier begins with one of the letters I, J, K, L, M, or N, the type is INTEGER.

- Any other first letter implies REAL.

The IMPLICIT statement may be used to alter the above conventions and/or introduce similar rules for types other than integer and real. See section 3.3.2.

For an asynchronous identifier, the first character after the dollar sign is used for implicit typing.

2.4 Constants

The type of a constant is determined by its form.

2.4.1 Integer Constants

An integer constant is a signed or unsigned string of decimal digits. It consists of up to 19 decimal digits in the range -2^{63} ($\approx -9.2 \times 10^{18}$) to $+2^{63}-1$ ($\approx 9.2 \times 10^{18}$).

The internal representation of an integer is a full word (64 bit) two's complement number. However, integer division produces a result accurate to only 46 bits.

EXAMPLES:

```
-1
1234567890
0
```

2.4.2 Real Constants

A real constant is a signed or unsigned string of decimal digits that includes a decimal point and/or an exponent. Any number of digits may be included, but only the first fifteen are significant. A real constant has one of the following forms:

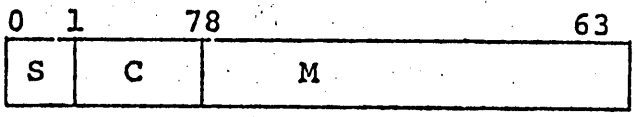
```
±i.          ±.i          ±i.i          ±iE±e
±i.E±e          ±.iE±e          ±i.iE±e
```

where i is a string of digits and e is a 1- or 2-digit exponent to the base of 10. The plus (+) character is optional. The magnitude of a real constant is in the approximate range of 5.4×10^{-79} to 7.2×10^{75}.

EXAMPLES:

```
3.1415
+0.031415E+2
.031415E2
31.415E-1
```

Internally, real data is stored as follows:



where S is the sign (+ = 0, - = 1) of the mantissa.
C is the base sixteen characteristic plus 64.
M is the mantissa (56 bits), represented as a sign-magnitude number.

The mantissa is hex-normalized.

2.4.3 Double Precision Constants

A double precision constant is a signed or unsigned string of decimal digits that includes an optional decimal point and an exponent. Any number of digits may be included, but only the first twenty nine are significant. A double precision constant has one of the following forms:

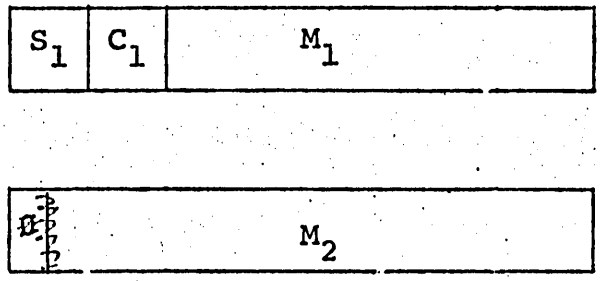
±i.iD±e ±i.D±e ±.iD±e ±iD±e

where i is a string of digits and e is a 1- or 2-digit exponent to the base 10. The plus (+) character is optional. The magnitude of a double precision constant is in the approximate range of 5.4×10^{-79} to 7.2×10^{75}.

EXAMPLES:

.31415927D1
3.1415D0
31415.93D-04

Internally, double precision constants appear as a real constant followed by an integer extension of the mantissa:



S_1 - Sign
C_1 - Characteristic
M_1 - Mantissa$_1$ (56 bits)
M_2 - Mantissa$_2$ (64 bits)

2.4.4 Complex Constants

A complex constant appears as an ordered pair of real constants enclosed in parentheses:

(r_1, r_2)

The real part is r_1 and the imaginary part is r_2.

EXAMPLES:

(0., -1.)
(5.2, 2.6)
(-3.E12, .017E-16)

Internally, complex data is stored as two real constants (see 2.4.2).

2.4.5 Logical Constants

A logical constant is a truth value.

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

A logical constant occupies one word which is either zero (.FALSE.) or non-zero (.TRUE.)

2.4.6 Hollerith Constants

A Hollerith constant is a string of characters which is represented internally by 8 bit ASCII codes. A Hollerith constant may appear wherever an expression may appear, however it may not appear within an expression. It is treated as an integer constant. There are four forms for Hollerith constants:

$nHC_1C_2...C_n$ $nRC_1C_2...C_n$

$'C_1C_2...C_n'$ $nLC_1C_2...C_n$

where n is an unsigned integer constant and C_i are characters.

Internally, the characters are packed eight per word. The two forms on the left are equivalent and imply left justification with trailing blanks (if necessary to pad n to a multiple of 8). The R and L forms indicate right and left justification respectively, however these two forms indicate binary zero padding (if necessary) not blanks. All characters including blanks, are significant in the string.

The apostrophe may be included in an apostrophe-delimited Hollerith constant by using two consecutive apostrophes in the string.

EXAMPLES:

	Storage	
1HA	A b b b b b b b	(ASCII)
2H A	b A b b b b b b	(ASCII)
'ABCDE'	A B C D E b b b	(ASCII)
'ABC''DEF'	A B C ' D E F b	(ASCII)
1RA	0000 0000 0000 0041	(HEX)
5LABCDE	4142 4344 4500 0000	(HEX)

2.4.7 Hexadecimal Constants

Hexadecimal constants are written as follows:

$$X'D_1D_2....'$$

where the D_i collectively represent a string of up to 16 hexadecimal digits (0-9, A-F). Hexadecimal constants are stored internally right justified in one word. They are treated as integers within expressions.

EXAMPLES:

```
X'FF'
X'000FF000'
```

3. SPECIFICATION STATEMENTS

Declarations are used to supply descriptive information about the program rather than to specify computation or other action. This descriptive information primarily concerns the interpretation of source program identifiers and object program storage allocation.

The following declaration statements must all appear in the program prior to any non-declarative statements and statement function definition statements.

Explicit type statements
IMPLICIT statement
DIMENSION statement
EXTERNAL statement
COMMON statement
EQUIVALENCE statement
MAXREGS statement

DATA statements are not required to appear in the program prior to any non-declarative statements.

3.1 Array Declarators

Arrays may be defined by several statements:

Explicit type statements
COMMON
DIMENSION

Array declarators are used in these statements. Each array declarator gives the array identifier and the maximum values each of its subscripts may assume, thus:

identifier (max_1, max_2, max_3)

The maxima must be integers. An array may have one, two, or three dimensions.

For example, the statement

```
DIMENSION EDGE (10,8)
```

specifies EDGE to be a two dimensional array whose first subscript may vary from 1 to 10 inclusive, and the second from 1 to 8 inclusive.

EXAMPLES:

```
DIMENSION PLACE (3,3,3) ,HI(2,4),K(256)
```

Arrays may also be declared in the COMMON and explicit type statements in the same way:

```
COMMON X(10,4), Y,Z
INTEGER A(7,32),B
DOUBLE PRECISION K(6,10)
```

3.1.1 Variable Dimensions

Within a subprogram, array declarations may use integer variables provided that the array name and variable dimensions are formal parameters of the subprogram. The actual array name and values for the dummy variables are given by the calling program when the subprogram is called.

Quantities needed for reference to the array are evaluated upon entry to the subprogram and unchanged by any subsequent modifications of the dimension variables.

EXAMPLE:

```
DIMENSION BETA (L, M, G)
```

The identifiers BETA, L, M, and G must all be formal parameters.

3.2 DIMENSION Statement

The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement.

FORM:

DIMENSION $S_1, S_2, \ldots, S_k$

where S is an array declarator.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in another statement. When the dimension information is provided in a COMMON or explicity type statement, it may not appear in a DIMENSION statement. The DIMENSION keyword may be abbreviated DIM.

3.3 TYPE Declarations

The explicit type statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL and the IMPLICIT statement are used to specify the type of the identifiers appearing in the program.

3.3.1 Explicit Type Statements

The general form of the explicit type statement is:

type $identifier_1$,$identifier_2$,...,$identifier_n$

Type may be INTEGER, REAL, DOUBLE, DOUBLE PRECISION, COMPLEX, or LOGICAL, and the $identifier_i$ are identifiers or array descriptors.

An identifier may appear in only one explicit type statement. Explicit type statements may be used to declare arrays that are not dimensioned in DIMENSION or COMMON statements.

3.3.1.1 INTEGER Statement

FORM:

```
INTEGER identifier,identifier,...
```

This statement declares the listed identifiers to be integer type with each datum occupying one word.

EXAMPLES:

```
INTEGER ALPHA,$PVAL
INTEGER TABLESIZE (10)
```

3.3.1.2 REAL Statement

FORM:

REAL identifier, identifier,...

This statement declares the listed identifiers to be real type with each datum occupying one word in floating point format.

EXAMPLES:

```
REAL $LOGX, MASS(10,4)
REAL I,J,K
```

3.3.1.3 DOUBLE PRECISION Statement

FORM:

DOUBLE PRECISION identifier, identifier,...

This statement declares the listed identifiers to be of double precision type. Each datum occupies two words in floating point format. The keyword PRECISION may be omitted.

EXAMPLES:

```
DOUBLE PRECISION RATE,Y,FLOW
DOUBLE $TIME(27,9)
```

3.3.1.4 COMPLEX Statement

FORM:

```
COMPLEX identifier,identifier,...
```

This statement declares the identifiers to be of complex type. Each datum occupies two words, two floating point numbers representing the real and imaginary parts.

EXAMPLE:

```
COMPLEX ZETA,W,ROOT
```

3.3.1.5 LOGICAL Statement

FORM:

LOGICAL identifier,identifier,...

This statement declares the listed identifiers to be of logical type. Each datum occupies one word where zero represents false and non-zero represents true.

EXAMPLE:

LOGICAL BOOL,$P,$Q,ANSWER

3.3.2 IMPLICIT Statement

The IMPLICIT statement is used to establish the implicit type of an identifier if the programmer wishes it to be different from the default implicit typing wherein identifiers beginning with letters I, J, K, L, M, N are integer, others are real.

FORMAT:

IMPLICIT typel $(A_1,A_2,...)$,type2$(A_3,A_4,...)$,...

where $type_i$ represents one of the following: INTEGER, REAL, DOUBLE, DOUBLE PRECISION, COMPLEX, LOGICAL; and $A_1,A_2,...$ represent single alphabetic characters or a range of characters (in alphabetic sequence) denoted by the first and last character of the range separated by a minus sign (e.g., A-D).

This statement causes any variable not mentioned in an explicit type statement and whose first character is one of those listed to be typed according to the type appearing before the list in which the character appears. For an asynchronous identifier, the first character after the dollar sign is used.

EXAMPLE:

```
IMPLICIT INTEGER (A-C,X),DOUBLE PRECISION(D),LOGICAL(L)
```

This statement would cause the following implicit declarations to be in effect:

1. Identifiers beginning with A, B, C, I, J, K, M, N, X are integer.
2. Identifiers beginning with D are double precision.
3. Identifiers beginning with L are logical.
4. Identifiers beginning with E, F, G, H, O, P, Q, R, S, T, U, V, W, Y, Z are real.

3.4 COMMON Statement

FORM:

COMMON block-list

or

REGISTER COMMON block-list

The COMMON statement specifies that certain variables or arrays are to be stored in an area also available to other programs. By means of COMMON statements, a program and its subprograms may share a common storage area. This area is located in data memory or register memory as indicated by the key word REGISTER. REGISTER COMMON may be abbreviated REGCOM.

The common area may be divided into separate blocks identifed by block names. A block is specified thus:

/identifier/identifier,identifier,...,identifier

The identifier enclosed in slashes is the block name. The identifiers which follow are the names of the variables or arrays assigned to the block. These elements are placed in the block in the order in which they appear in the block specification. A normal common and a register common may not have the same block name.

The block list of the COMMON statement consists of a sequence of one or more block specifications. For example the statement

```
COMMON/R/X,Y,T/C/U,V,W,Z
```

indicates that the elements X,Y, and T, in that order, are to be placed in block R and that U,V,W,Z are to be placed in block C.

Block entries concatenate throughout the program, beginning with the first COMMON statement. For example the statements

```
REGISTER COMMON/D/ALPHA/R/A,B/C/S
REGISTER COMMON/C/X,Y/R/U,V,W
```

have the same effect as the statement

```
REGISTER COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of common storage may be left unlabeled and is called blank common. Blank common is indicated by two consecutive slashes. A blank common may not be declared to be of type REGISTER.

For instance

```
COMMON/R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank common.

The slashes may be omitted when blank common is the first block of the statement.

```
COMMON B,C,D
```

Storage allocation for blocks with the same name begins at the same location for all programs executed together. For example if a program contains

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has

```
COMMON /R/U,V,W//D,E,F
```

as its first COMMON statement, then the quantities represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

Labeled blocks of a given name must have the same length in all programs executed together.

Blank common may be any length in any program.

Array names appearing in COMMON statements may have dimension information appended, as in a DIMENSION statement. For example

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common.

3.5 EQUIVALENCE Statement

The EQUIVALENCE statement allows more than one identifier to represent the same quantity.

FORM:

EQUIVALENCE(R_1,R_2,...),(R_N,R_{N+1},...),...

where R_i is a reference. (EQUIVALENCE may be abbreviated EQUIV.)

The references of an EQUIVALENCE statement may be simple variables or array identifiers or array element references. The subscripts of an array element must be integer constants. The number of subscripts must be equal to the array dimension or must be one. Synchronous and asynchronous variables may not be equivalenced to each other.

EXAMPLE:

```
EQUIVALENCE(A,B,C(3)),(T(4),S(1,1,2))
```

The inclusion of two or more references in a parenthesis pair indicates that the quantities referenced are to share same memory locations. For example

```
EQUIVALENCE(RED,BLUE)
```

specifies that the quantities RED and BLUE are stored in the same place.

When no array subscript is given, it is taken to be 1, thus

```
EQUIVALENCE(X,Y)
```

is the same as

```
EQUIVALENCE(X,Y(1))
```

Elements of multiply dimensioned arrays may be referenced with a single subscript by use of the element successor function. For example in the three-dimensional array specified by

```
DIMENSION ALPHA (N1,N2,N3)
```

the position of element ALPHA (K_1,K_2,K_3) is given by

element position = $(K_3-1)*N_1*N_2+(K_2-1)*N_1+K_1$

Thus the sequence

```
DIMENSION BETA(4),ALPHA(2,3,4)
EQUIVALENCE(BETA(2),ALPHA(8))
```

specifies that BETA(2) and ALPHA(2,1,2) are stored in the same place.

Since the entire arrays are shifted to satisfy the equivalence only the relative positions of the references are important. In the examples below

```
EQUIVALENCE(BETA(1),ALPHA(7))
```

or

```
EQUIVALENCE(BETA,ALPHA(7))
```

will do as well.

Note that the relation of equivalence is transitive, e.g., the two statements.

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

have the same effect.

3.5.1 EQUIVALENCE and COMMON

Identifiers may appear in both COMMON and EQUIVALENCE statements provided the following rules are observed.

No two quantities in common may be set equivalent to one another.

Quantities placed in a common block by means of equivalences may cause the end of the common block to be extended. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
DQUIVALENCE(A,Y)
```

causes the common block R to extend from X to A(4), arranged as follows:

```
X
Y A(1)
Z A(2)
  A(3)
  A(4)
```

Equivalence which cause extension of the start of a common block are not allowed. For example the sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (X,A(3))
```

is not permitted since it required block R to be arranged

```
    A(1)
    A(2)
  X A(3)
  Y A(4)
  X
```

A(1) and A(2) extend the start of block R.

3.6 EXTERNAL Statement

FORM:

EXTERNAL identifier,identifier,...,identifier

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must appear in an EXTERNAL declaration in the calling program.

EXAMPLE:

```
      EXTERNAL SIN,COS
               .
               .
               .
      CALL TRIGF(SIN,1.5,ANSWER)
               .
               .
               .
      CALL TRIGF(COS,.87,ANSWER)
               .
               .
               .
      END
      SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
               .
               .
               .
      ANSWER = FUNC(ARG)
               .
               .
               .
      RETURN
      END
```

3.7 MAXREGS Statement

FORM:

```
MAXREGS    ic
```

where ic is an integer constant.

This statement tells the compiler to limit its use of registers in the generated object code to the value specified by ic.

There is a minimum number of registers required in each module. This lower bound is unkown at present. If ic is less than this number, a warning message is generated and the value of ic is reset to this lower bound.

3.8 DATA Specification

The data specification statements DATA, CONST, and BLOCK DATA are used to declare constants and specify initial values for variables. These values are compiled into the object program. They become the values assumed by the variables when execution begins.

3.8.1 DATA Statement

The data to be compiled into the object program is specified in a DATA statement.

FORM:

```
DATA v/d/,v/d/,...
```

where v is a variable list and d is a data list.

The variable lists in a DATA statement consist of simple variables, array names, or array elements separated by commas.

Variables in common may appear on the lists only if the DATA statement occurs in a BLOCK DATA subprogram. Asynchronous variables may appear in a DATA statement.

The data items of each data list correspond one-to-one with the variables of each variable list. Each data item specifies the value given to its corresponding variable.

Data items may be numerical constants or alphanumeric strings. For example

```
DATA ALPHA,BETA/5,16.E-2/
```

specifies the value 5 for ALPHA and the value .16 for BETA.

Any data item may be preceded by an integer followed by an asterisk. The integer indicates the number of times the item is to be repeated. For example

```
DATA A(1),A(2),A(3),A(4),A(5)/61E2,4*32E1/
```

specifies 5 values for the array A; the value 6100 for A(1) and the value 320 for A(2) through A(5).

When an unsubscripted array name is included in the variable list it implies that all elements of the array are to be initialized. It is equivalent to writing out all elements of the array in sequence.

EXAMPLE:

The above example could be written:

```
DIMENSION A(5)
DATA A/61E2,4*32E1/
```

The form of the constant, rather than the type of the variable, determines the data type of the stored constant.

Hollerith constants, or alphanumeric strings, are treated specially in the DATA statement. A single Hollerith constant may initialize more than one variable element, whereas any other type of constant corresponds to a single element. A Hollerith constant may not contain excess characters, but it will be extended with blanks so as to fill an integral number of elements.

EXAMPLES:

```
DIMENSION I(2)
DATA I/'A','B'/

implies    I(1) = 'A   '
           I(2) = 'B   '
DIMENSION I(2)
DATA I/'ABCDEFGHIJK'/

implies    I(1) = 'ABCDEFGH'
           I(2) = 'IJK     '

DATA J/'ABCDEFGHIJK'/

is an error
DIMENSION R(2)

DATA R/2*'A'/
implies    R(1) = 'A   '
           R(2) = 'A   '
```

3.8.2 CONST Statement

The CONST statement has the same form and general meaning as the DATA statement.

FORM:

CONST v/d/,v/d/,...

where v is a variable list and d is a data list. The syntax and semantics of these lists are identical with those of the DATA Statement with one exception: The CONST statement causes the variable to be allocated in constant memory. This implies that they are in fact no longer variables, rather, symbolic constants. Hence, they may not be the destination of an assignment statement or a READ statement. Also, they may not be asynchronous.

3.8.3 BLOCK DATA Statement

FORM:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data specification for variables in common blocks require the use of a BLOCK DATA subprogram.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only declarative statements associated with the data being defined.

EXAMPLE:

```
BLOCK DATA
COMMON /R/X,Y/C/Z,W,V
DIMENSION Y(3)
COMPLEX Z
DOUBLE PRECISION X
DATA Y (1),Y(2),Y(3),/1E-1,2*3E2/
DATA X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one block of common in one subprogram. However, any common block mentioned must be listed in full. In the example above W and V are listed in block C although no data values are defined for them.

3.9 PRIVATE Statement

FORM:

PRIVATE identifier,identifier,...,identifier

This statement, like the EXTERNAL statement, declares the listed identifiers to be subprogram names. However it further requires that the listed subprograms must not be shared with any other program units. Hence, if two program units referenced subprogram SUBR and at least one of them states PRIVATE SUBR; two copies of SUBR will be loaded by the linker, one for each program unit.

NOTE: The replication applies to ~~REGISTER~~ register memory, ~~CONSTANT~~ constant memory, and data memory, but not to program memory.

4. EXPRESSIONS

Expressions are strings of operands separated by operators. There are two types of FORTRAN expressions: arithmetic and logical.

4.1 Arithmetic Expressions

An arithmetic expression is a sequence of basic elements separated by arithmetic operators and parentheses in accordance with mathematical convention and the rules given below. An arithmetic expression has a single numerical value,namely, the result of the calculations specified by the quantities and operators comprising the expression.

The arithmetic operators are +,-,*,/,**, denoting, respectively, addition, subtraction, multiplication, division and exponentiation.

An expression may consist of a single element (constant, variable, or function reference):

```
2.71828
Z(N)
TAN(THETA)
```

Compound expressions may be formed by using operators to combine basic elements:

```
X+3
TOTAL/POINTS
TAN(PI*M)
```

Any expression may be enclosed in parentheses and considered to be a basic element:

```
(X+Y)/2
(ZETA)
COS(SIN(PI*M)+X)
```

Expressions may be preceded by a + or - sign:

```
+X
-(ALPHA BETA)
-SQRT(-GAMMA)
```

However, no two operators may appear in sequence, for instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

By repeated use of the rules above, all permissible arithmetic expressions may be formed.

4.1.1 Operator Precedence

If the precedence of operations is not given explicitly by parentheses, it is understood to be the following (in order of decreasing precedence):

OPERATOR	OPERATION
**	Exponentiation
* and /	Multiplication and Division
+ and -	Addition and Subtraction or Negatic

For example,the expression

```
A*B+C/D**E
```

is taken to be

```
(A*B)+(C/(D**E))
```

Sequences of operations of equal precedence are performed left to right except for exponentiation which is performed right to left.

EXAMPLES:

W*X/Y/7	is evaluated as ((W*X)/Y)/7
A**B**C	is evaluated as A**(B**C)

4.1.2 Evaluation of Mixed-Type Expressions

The value of a arithmetic expression may be of integer, real, double precision, or complex type. The type of the expression is determined by the types of its elements according to the rules which follow.

The arithmetic types are ranked as follows:

RANK	TYPE
1	Integer
2	Real
3	Double Precision
4	Complex

The type of an expression is the type of the highest ranking element in the expression.

Each operation within an expression is evaluated in the type of the operations highest ranking operand. Thus the evaluation of an expression is not changed to a higher rank until necessary.

EXAMPLE:

I/J+R*DP*C

is evaluated as

CMPLX(FLOAT(I/J),0.)+CMPLX(REAL(DBLE(R)*DP),0.) * C

~~CMPLX(REAL(DBLE(FLOAT(I/J)+R)*DP),0.)*C~~

Integer expressions are evaluated using binary integer arithmetic throughout. In integer arithmetic fractional parts arising in division are truncated, not rounded. For example:

7/3 yields 2; 3/7 yields 0

All other calculations use binary floating point arithmetic.

Conversions to higher rank are performed as follows:

1. An integer quantity becomes the integer part of a real quantity. The fractional part is zero.
2. A real quantity becomes the most significant part of a double precision real quantity. The least significant part is zero.
3. A real quantity becomes the real part of a complex quantity. The imaginary part is zero.
4. A double precision quantity is converted to single precision and becomes the real part of a complex quantity. The imaginary part is zero.

4.2 Logical Expressions

There are four basic elements used in FORTRAN logical expressions: logical constants, logical type variables, logical type functional references, and relations. All of these basic elements represent logical quantities.

A logical quantity may have either of two values: true or false. Logical quantities occupy one word of memory.

4.2.1 Relations

Relations are constructed from numerical expressions of integer, real, or double precision type through the use of relational operators. The relational operators are:

OPERATOR	RELATION
.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 periods are part of the operator and must be present.

Two expressions of integer, real, or double precision type separated by a relational operator form a relation. For example

```
X+2.LE.3*Y
```

is a relation. The entire relation constitutes a basic logical element.

The value of such an element is true if the relation expressed is true and false otherwise. In the example above, the element has the value true if X is 2 and Y is 2, and the value false if X is 2 and Y is 1.

Complex operands are allowed for the operators .EQ. and .NE. only.

Relational operators have lower precedence than arithmetic operators.

4.2.2 Logical Operators

The logical operators are .NOT., .AND. and .OR. denoting, respectively, logical negation, logical multiplication and logical addition. The enclosing periods are part of the operators and must be present. The logical operators are defined as follows (where P and Q are logical expressions);

.NOT.P	true if P is false, false if P is true
P.AND.Q	true if P and Q are both true, otherwise false
P.OR.Q	false if P and Q are both false, otherwise true

A logical expression may consist of a single logical element. For example:

```
.TRUE.
BOOL(N)
X.GE.3.14159
```

Single elements may be combined through use of the logical operators .AND. and .OR. to form compound expressions, such as:

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.LIMIT
```

Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.OR.S).AND.(R.OR.Q)
(BOOL(M))
PARITY((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

Any logical expression may be preceded by the operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

By repeated use of these rules all permissible logical expressions may be formed.

When the precedence of operations is not given by parentheses, it is understood to be the following (in decreasing order of precedence):

OPERATOR	OPERATION
.NOT.	logical negation
.AND.	logical multiplication
.OR.	logical addition

Thus the expression

```
T.AND..NOT.S.OR..NOT.P.AND.R
```

is interpreted

```
(T.AND.(.NOT.S)).OR.((.NOT.P).AND.R)
```

4.3 Summary of Operator Precedence

When the precedence of operators is not given explicitly by parentheses, it is understood to be as follows (in order of decreasing precedence):

OPERATOR	OPERATION
**	exponential
*,/	multiply, divide
+,-	add, subtract, negate
.GT.,.GE.,.LT., .LE.,.EQ.,.NE.	relational
.NOT.	logical negation
.AND.	logical multiply
.OR.	logical add

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY.AND.
(X.EQ.Y))
```

5. ASSIGNMENT STATEMENTS

Assignment statements are the basic executable statements of the FORTRAN language. Two types of assignment statements are available: arithmetic and logical.

5.1 Arithmetic Assignment Statement

The arithmetic assignment statement specifies an arithmetic expression to be evaluated and a variable to which the expression value is to be assigned.

FORM:

```
variable = expression
```

The character "=" is an operational symbol signifying replacement, not equality. Thus the first example below means "take the current value of Y, double it, and assign the result to Y."

EXAMPLES:

```
Y = 2*Y
A = -A
X(N) = N*ZETA(ALPHA)*(M/PI)+(1.0,-1.0)
```

Type conversion is provided if the variable is of a type different from the expression.

5.2 Mixed-Type Assignment

Mixed-type assignment involves arithmetic assignment statements in which the type of the expression on the right differs from the type of the variable on the left. The evaluated expression is converted to the type of the variable. The following table gives the conversions for all combinations of expressions and variables.

Type of Variable	Type of Expression	Conversion
INTEGER	INTEGER	None.
INTEGER	REAL	Truncate fractional part, convert to integer form.
INTEGER	DOUBLE PRECISION	Truncate fractional part, convert to integer form.
INTEGER	COMPLEX	Truncate fractional part of real part, convert to integer form.
REAL	INTEGER	Convert to floating-point form.
REAL	REAL	None.
REAL	DOUBLE PRECISION	Take first word of extended floating point.
REAL	COMPLEX	Take real part.
DOUBLE PRECISION	INTEGER	Convert to extended floating point form.
DOUBLE PRECISION	REAL	Extend single floating point to extended form.
DOUBLE PRECISION	DOUBLE PRECISION	None.
DOUBLE PRECISION	COMPLEX	Extend real part to extended floating point form.
COMPLEX	INTEGER	Convert to real form, assign to real part, assign zero imaginary.
COMPLEX	REAL	Assign to real part, assign zero imaginary.
COMPLEX	DOUBLE PRECISION	Take first word of extended floating point, assign to real part, assign zero imaginary.
COMPLEX	COMPLEX	None.

Assignment Statements of the form

variable = Hollerith constant

are special cased. The character string represented by the Hollerith constant is transferred to the variable without any type conversion. The string is stored as indicated in section 2.4.6 The number of characters cannot be greater than the number of bytes in the variable.

EXAMPLE:

```
X = 'AB'
```

implies the eight bytes represented by X will appear as

A	B	blank	blank	blank	blank	blank	blank

5.3 Logical Assignment Statement

The logical assignment statement specifies a logical expression to be evaluated and logical variable to which the expression value to be assigned.

FORM:

variable = expression

Both the variable and expression must have logical type.

EXAMPLES:

```
LOGICAL LA,LB,LC,LD
LA = LB.AND.LC.AND.LD
LB = .NOT.LA
LC = A.GT.B.OR.C.EQ.D
```

6. CONTROL STATEMENTS

The normal flow of a FORTRAN program is sequentially through the statements in the order in which they are given to the compiler. By means of control statements, the programmer may specify the flow of the program.

6.1 Statement Numbers

FORTRAN statements may be given numbers to be referenced by control statements. A statement number is written as an unsigned integer of five digits or less. Leading zeroes and embedded blanks are ignored.

Although statement numbers are written as integers, they do not represent numerical quantities. Statement numbers represent statement labels, a distinct basic quantity. Statement numbers are used for program control, not numerical calculation.

Statement numbers must be unique, i.e., no two statements may have the same number.

6.2 GO TO Statements

GO TO statements unconditionally transfer control from one part of the program to another. There are three forms of the GO TO statement: unconditional, computed, and assigned.

6.2.1 Unconditional GO TO Statement

FORM:

```
GO TO n
```

where n is a statement number.

This statement transfers control to the statement numbered n.

EXAMPLE:

```
GO TO 345
```

6.2.2 Computed GO TO Statement

FORM:

GO TO $(n_1,n_2,\ldots,n_k)$,variable

where $n_1,n_2,\ldots,n_k$ are statement numbers.

The variable must be of integer type. This statement transfers control to the statement numbered $n_1,n_2,\ldots n_k$ if the variable has the value 1,2,...k, respectively. Values outside the range 1 through k cause transfer of control to the following statement. The comma preceding the variable may be omitted.

EXAMPLES:

```
GO TO (22,3,7),SWITCH
GO TO (1,2,62,78) Y
```

6.2.3 Assigned GO TO Statements

FORMS:

GO TO variable

GO TO variable,(n_1,n_2,...,n_k)

where n_1,n_2,...n_k are statement numbers.

The second form is allowed for compatibility only; the labels n_1,...n_k are not used. The comma following the variable may be omitted.

The variable must be a scalar of integer type and may not be an asynchronous variable. This statement transfers control to the statement whose number was last assigned to the variable. The assignment must take place in a previously executed ASSIGN statement.

The variable is a control variable, having a label as a value, not a numerical quantity.

EXAMPLES:

```
GO TO ERROR
GO TO X(100,200)
```

6.2.4 ASSIGN Statement

FORM:

ASSIGN statement number TO variable

The variable must be a scalar of integer type and may not be an asynchronous variable. This statement assigns the value of the variable for a subsequent assigned GO TO statement. The statement number represents the statement to which the assigned GO TO will transfer control.

EXAMPLES:

```
ASSIGN 7 TO LABEL
ASSIGN 13 TO ERROR
```

6.3 Arithmetic IF Statement

FORM:

IF (expression)n_1,n_2,n_3

where n_1,n_2,n_3 are statement numbers.

This statement transfers control to the statement numbered n_1,n_2 or n_3 if the value of the expression is less than, equal to, or greater than zero, respectively. The expression must be of integer, real, or double precision type.

EXAMPLES:

```
IF (ETA)4,7,12
IF (KAPPA-L(10))20,14,14
```

6.4 Logical IF Statement

FORM:

IF (expression)S

where S is a complete statement.

The expression must be a logical expression. S may be any imperative (executable) statement other than a DO statement or another logical IF statement.

If the value of the expression is false, then control passes to the next sequential statement.

If the value of the expression is true, statement S is executed. After execution of S, control passes to the next sequential statement unless S is an arithmetic IF statement or GO TO type statement in which case control is transferred as indicated.

As an example, consider the statements

```
IF(B)Y=X*SIN(Z)
W = Y**2
```

If the value of B is true the statements Y=X*SIN(Z) and W = Y**2 are executed in that order. If the value of B is false, the statement Y=X*SIN(Z) is not executed.

EXAMPLES:

```
IF(T.OR.S)X=Y+1
IF(Z.GT.X(K))CALL SWITCH(X,Y)
IF(K.EQ.INDEX)GO TO 15
```

6.5 DO Statement

FORMS:

```
DO n index = initial, limit
DO n index = initial, limit, step
```

where n is a statement number.

The index must be a simple integer variable, and may not be an asynchronous variable. The initial and limit may be integer simple variables, signed integer constants, or integer expressions. The step must be an integer simple variable, an integer constant or an integer expression. If the step is not given, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. Initially, the initial value is assigned to the index. Thereafter, after each execution of the range, the step value is added to the index value and the result assigned to the index.

Prior to each execution of the range, the index value is compared with the limit value. If the index value does not exceed the limit value, the range is executed. This differs from Standard FORTRAN where the range is always executed once before the first test.

After the last execution of the range, control passes to the statement immediately following the range. This exit from the range is called the normal exit. Exit may also be accomplished by a transfer from within the range.

The range of a DO statement may include other DO statements provided that the range of each contained DO statement is entirely within the range of the containing DO statement.

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer exit from the range, the index retains its current value and is available for use as an ordinary variable. After a normal exit from the range, the index has the value which caused the exit.

The values of the index, limit, and step may be altered within the range of the DO statement. altering the values of limit or step will not affect the loop.

The range of a DO statement must not end with a GO TO type statement or an arithmetic IF statement. A logical IF statement is allowed as the last statement of the range, provided the logical IF does not contain a GO TO type statement or an arithmetic IF. In this case, control is transferred thus:

> The range is considered ended when and if control would normally pass to the statement following the logical IF statement.

As an example, consider the sequence:

```
DO 5K = 1,4
5 IF(X(K).GT.Y (K))Y(K)=X(K)
6...
```

Statement 5 is executed four times, whether the statement Y(K) = X(K) is executed or not. Statement 6 is not executed until statement 5 has been executed four times.

EXAMPLES:

```
DO 22 L = 1,30
DO 45 K = 2,LIMIT,3
```

6.6 CONTINUE Statement

FORM:

```
CONTINUE
```

This statement is a dummy statement, used primarily as a target point for transfers, particularly as the last statement in the range of a DO statement. For example in the sequence

```
      DO 7K = START,END
      .
      .
      .
      IF(X(K))22,13,7
    7 CONTINUE
```

a positive value X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

6.7 PAUSE Statement

FORMS:

```
PAUSE
PAUSE n
PAUSE 'character string'
```

where n is an integer constant.

This statement causes the operand to be displayed on the system output device. If no operand is given, zero is displayed.

EXAMPLE:

```
PAUSE 167
```

6.8 STOP Statement

FORMS:

```
STOP
STOP n
STOP 'character string'
```

where n is an integer constant.

This statement terminates the program. It does not stop the machine and may be used to transfer control to the operating system.

6.9 PURGE Statement

FORM:

PURGE $N_1, N_2, \ldots N_n$

where the N_i are asynchronous variables.

This statements causes all the named variables to be made empty, regardless of their former state.

7. INPUT/OUTPUT

Input/output statements specify the transfer of information between computer memory and input/output devices, or between one part of computer memory and another. These statements also allow the program to manipulate I/O devices.

Information may be transferred in two different forms: formatted and unformatted. The unformatted form involves no data conversion, data is transferred in its internal format. Formatted data is converted from internal to external form, or vice versa, under control of a FORMAT specification in the program.

7.1 Input/Output Lists

Input/output statements may contain a list of variables which are to receive values on input or are to provide values for output. The list of a transmission statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example

```
READ (1,3)L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by the index control, is enclosed in parentheses and the whole acts as a single element of the list.

For example

```
READ(7,23) (X(K),K=1,4)
```

is equivalent to

```
READ(7,23) X(1),X(2),X(3),X(4)
```

The initial, limit, and step values are given as in the DO statement:

```
READ(4,2) N,(GAIN(K),K=1,M,N)
```

The indexing may be compounded as in the following

```
READ (1,13) ((MASS(K,L)K=1,5),L=1,4)
```

This statement reads in the elements of array MASS in the order
MASS(1,1),MASS(2,1),...,MASS(5,1),MASS(1,2),...,MASS(5,4)

If an entire array is to be transmitted, the indexing may be omitted and only the array identifier written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus the example above can be written

```
READ(1,13)MASS
```

7.2 Input/Output Statement Parameters

Many input/output statements have similar formats. The following definitions apply to all input/output statements in which they may appear:

u Logical I/O unit number which may be an unsigned integer constant or an integer simple variable and may not be an asynchronous variable. The correspondence between unit number and actual I/O device is determined by machine configuration and operating system.

f Format declaration identifier which may be the statement number of a FORMAT statement in the program or an array name where the format specification is to be found. f may not be an asynchronous variable.

list An I/O list as defined in section 7.1.

S_1 Statement number to which program control is transferred in the event of detecting an end-of-file indication on the I/O unit while processing the statement.

S_2 Statement number to which program control is transferred if any error is detected during the processing of the statement.

7.3 Sequential Input/Output Statements

These statements treat input and output to and from I/O units as if the units contained sequential files each composed of an ordered set of records. Each time a READ or WRITE statement is executed, at least one record is processed. And as each record is processed, the file is positioned to read or write the next sequential record.

7.3.1 READ Statement

FORMS:

READ (u,f,END=S_1,ERR=S_2)list

READ (u,END=S_1,ERR=S_2)list

where u is an I/O unit designation and f is a format reference.

The parameters END=S_1 and ERR=S_2 are optional and their order may be reversed.

The READ statement causes information to be read from the I/O unit designated and stored in memory as values of the variables in the list.

In transmission of formatted data, the conversion from external to internal form is specified by the format referenced (first form).

When binary data is transmitted the format reference is omitted (second form).

EXAMPLES:

```
READ(1,15 END=100)ETA,PI
READ(K+L,10)GAIN,ZAI
READ(M,FMT,ERR=999, END=100)(TABL(K),K=1,M)
READ(TAPE)(TEMP(L),L=1,100)
```

7.3.2 WRITE Statement

FORMS:

```
WRITE(u,f,END=S1,ERR=S2)list
WRITE(u,END=S1,ERR=S2)list
```

where u is an I/O unit designation and f is a format reference.

The END=S_1 and ERR=S_2 parameters are optional and their order may be reversed.

The WRITE statement causes the values of the variables of the list to be transmitted from memory to the designated I/O unit.

In transmission of formatted data, the conversion from internal to external form is specified by the format referenced (first form).

When binary data is transmitted the format reference is omitted (second form).

```
EXAMPLES:
WRITE(2,15)ZILCH
WRITE(K3,4,)(A(K),K=2,20),ICHI
WRITE(N1,CONV)H1,H2
WRITE(4)NUMB,(SYMB(J),J=1,NUMB)
```

7.4 Internal Transmission

The ENCODE and DECODE statements are similar to formatted READ and WRITE statement except that no I/O unit is used in the data transfer. Data is transferred under format specifications from one area of computer memory to another.

7.4.1 ENCODE Statement

FORM:

```
ENCODE(c,f,b,n)list
      or
ENCODE(c,f,b)list
```

where c is an integer constant or integer variable describing the number of characters per record in storage.

f is a format reference.

b is a simple variable, array reference, or array name at which the first record is to start. This is the "buffer."

n is a simple integer variable into which the number of characters actually processed will be stored. n may not be an asynchronous variable.

list as defined for I/O list.

This statement converts the data in the list according to the format and stores it in records beginning at b, with c characters per record. If the format attempts to convert more than c characters per record, a diagnostic is given. If the format converts less than c characters, the remainder of the record is filled with blanks.

EXAMPLE:

```
ENCODE(80,100,BUFF)A,B,C
ENCODE(I,FMT,B,COUNT)D,Z(J)
```

7.4.2 DECODE Statement

FORM:

```
DECODE(c,f,b,n)list
      or
DECODE(c,f,b)list
```

where c,f,b,n are as defined for ENCODE.

This statement converts and edits the data from the records starting at b and consisting of c characters each, and stores it in the variables specified in the list. When the format specifies more than c characters per record, a diagnostic is given. When fewer than c characters per record are specified, the remainder of the record is ignored.

EXAMPLE:

```
DECODE(78,103,BUFFER)LDATA
```

7.5 FORMAT Specifications

The format designator f appearing in formatted I/O statements may be the name of an array containing a format string or it may be the statement label of a FORMAT statement.

FORM:

FORMAT(S_1,S_2,...,S_k)

where S is a data field specification.

7.5.1 Numeric Fields

Conversion of numerical data may be one of six types.

1. type - D
   internal form -- binary floating point (double precision)
   external form -- decimal floating point (double precision)

2. type - E
   internal form -- binary floating point
   external form -- decimal floating point

3. type - F
   internal form -- binary floating point
   external form -- decimal fixed point

4. type - G
   internal form -- binary floating point
   external form -- decimal fixed point or floating point

5. type - I

internal form -- binary integer
external form -- decimal integer

6. type - Z

internal form -- binary integer
external form -- hexadecimal integer

These types of conversion are specified by the forms

1. Dw.d
2. Ew.d
3. Fw.d
4. Gw.d
5. Iw
6. Zw

respectively. The letter D,E,F,G,I, or Z designates the conversion type; w is an integer specifying the field width; d is an integer specifying the number of decimal places to the right of the decimal point. As an aid in conversion, the letter O will be allowed and treated as a Z. For example, the statement

```
FORMAT(I5,F10.2,D25.15)
```

could be used to output the line

```
32    -17.60    5.9625478777541D03
```

on the output listing.

The type of conversion used should correspond to the type of the variable in the input-output list. I conversion is used for integer type variables, E, F, or G conversion is used for real type variables, and D conversion is used for double precision type variables.

The decimal fixed point number (type F) has a decimal point but no exponent, whereas the decimal floating point (type E or D) has an exponent. On output, the exponent always has the form shown, i.e., and "E" or "D" followed by a signed, two digit integer. On input, however, the "E", "D", or the "+" sign, or the entire exponent may be omitted on the external form. For example, the following are all valid E15.6 fields:

```
  .317250+2
  .317250E2
  .042739-45
31064
```

The field width w includes all of the characters (decimal point, signs, blanks, etc.) which comprise the number. If a number is too long for its specified field, the excess characters are lost. Since numbers are right justified in their fields, the loss is from the most significant part of the number.

During input, the appearance of a decimal point "." in an E, D, or F type number overrides the d specification of the field. In the absence of an explicit decimal point, the point is positioned d places from the right of the field, not counting the exponent, if present. For example, a number with external appearance 271828E-1 and specification E12.5 is interpreted as 2.71828E-1.

7.5.2 Scale Factors

Scale factors may be specified for D, E, and F type conversions. A scale factor is written nP where P is the identifying character and n is a signed or unsigned integer specifying the scale factor.

For F type conversion the scale factor specifies a power of ten such that

external number = (internal number)*(power of ten)

For D and E type conversions, the scale factor multiplies the number by a power of ten but the exponent is changed accordingly, leaving the number unchanged except in form. For example if the statement

```
FORMAT(F8.3,E16.5)
```

corresponds to the line

```
26.451          -4.1321E-02
```

then the statement

```
FORMAT(-1PF8.3,2PE16.5)
```

corresponds to the line

```
2.645           -41.32100E-03
```

When no scale factor is given, it is understood to be 0 for F and 1 for D and E. However, once a scale factor is given, it holds for all following D, E, and F type conversions within the same format. The scale factor is reset to zero or one by giving a scale factor of zero or one, respectively.

Scale factors have no effect on I conversions.

7.5.3 G-Fields

Output under control of a G-field is dependent on the magnitude of the floating point number being converted. Where m represents the magnitude of the number, the following table shows the relationship between m and the conversion field to be used.

Magnitude	Conversion Field
$0.1 \leq m \leq 1$	F(w-4).d, 4X
$1 \leq m \leq 10$	F(w-4).(d-1),4X
.	
.	
.	
$10^{d-2} \leq m \leq 10^{d-1}$	F(w-4).1, 4X
$10^{d-1} \leq m \leq 10^{d}$	F(w-4).0, 4X
$m \leq .1$ or $m \geq 10^{d}$	sEw.d

s is the current scale factor and applies only when the E conversion field is used, 4X denotes a field of four spaces.

Input under control of a G-field is the same as for the F-field.

7.5.4 Logical Fields

Logical data can be transmitted in a manner similar to numeric data by use of the form:

Lw

where L is the control character and w is an integer specifying the field width.

Data is transmitted as the value of a logical variable in the input-output list.

On input, the data field is inspected for a T or F, if one is found the value of the logical variable will be stored as true or false, respectively. If the data field contains no T or F, a value of false will be stored.

On output, w-1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

7.5.5 Alphanumeric Fields

Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form Aw or Rw; A and R are the control characters and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input-output list. The variable may be of any type. For example, the sequence

```
      READ (2,5)V
    5 FORMAT (A4)
```

causes four characters to be read and placed in memory as the value of the variable V.

The character information is transferred as 8 bit ASCII characters, stored 8 characters per 64 bit word.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type. If w exceeds the maximum, leading characters are lost on input and replaced with blanks on output. When w is less than the maximum, the A format causes left justification with blank fill on input and only the left-most w characters are used for output. The R format causes right justification with binary zero fill on input and only the right-most w characters are used for output.

7.5.6 Alphanumeric Constant Fields

An alphanumeric constant may be specified within a format by preceding the alphanumeric string by the form nH. H is the control character and n is the number of characters in the string, counting blanks. For example, the statement

```
FORMAT(17H PROGRAM COMPLETE)
```

can be used to output

```
PROGRAM COMPLETE
```

on the output listing.

Apostrophe delimited alphanumeric strings may be used in the same manner.

7.5.7 Mixed Fields

An alphanumeric format field may be placed among other fields of the format. For example, the statement

```
FORMAT(I5,8H FORCE = F10.5)
```

can be used to output the line

```
22 FORCE = 17.68901
```

Note that the separating comma may be omitted after an alphanumeric format field.

7.5.8 Blank or Skip Fields

Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X and n is the number of blanks or characters skipped. n must be greater than zero.

For example, the statement

```
FORMAT(5H STEPI5,10X,3HY = F7.3)
```

may be used to output the line

```
STEP 28        Y = -3.872
```

where ten blanks separate the two quantities.

7.5.9 Tabulation

The position in the record where the transfer of data is to begin can be specified by T format conversion. The specification is Tn where n is the character position. For printed output, the first character is for carriage control and should not be counted.

EXAMPLE:

```
FORMAT(T20,'NAME',T40,'AGE',T1,6H GRADE)
```

would print a line:

```
Position 1          Position 19          Position 39
    ↓                   ↓                    ↓
 GRADE                NAME                 AGE
```

7.5.10 Repetition of Field Specifications

Repetition of a field specification may be specified by preceding the control character D, E, F, G, I by an unsigned integer giving the number of repetitions desired. For example

```
FORMAT(2E12.4,3I5)
```

is equivalent to

```
FORMAT(E12.4,E12.4,I5,I5,I5)
```

7.5.11 Repetition of Groups

A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number.

For example

```
FORMAT(2I8,2(EI5.5,2(F8.3)))
```

is equivalent to

```
FORMAT(2I8,E15.5,2F8.3,E15.5,2F8.3)
```

Up to nine levels of parentheses are allowed in group repetition, in addition to those enclosing the entire format. For example, the statement

```
FORMAT(I5,2(E15.5,2(F8.3)))
```

~~exhibits the maximum degree of nesting.~~ The outermost parenthesis pair is termed level zero, the next level one, and the innermost level two.

7.5.12 Complex Fields

Complex quantities are transmitted as two independent real quantities. The format specification is given as two successive real specifications or one repeated real specification. For instance, the statement

```
FORMAT(2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

7.5.13 Multiple-Record Formats

To handle a group of input-output records where different records have different field specifications, a slash "/" is used to indicate a new record. For example, the statement

```
FORMAT(3I8/I5,2F8.4)
```

is equivalent to

```
FORMAT(3I8)
```

for the first record and

```
FORMAT(I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used.

Blank records may be written on output or records skipped on input by using consecutive slashes.

Both the slash and the closing parentheses at the end of the format indicate the termination of a record. If the list of an input-output statement dictates that transmission of data is to continue after the closing parentheses of the format is reached, the format is repeated from the last open parentheses level of one or zero. If this parenthesis is preceded by a repeat specification, the repeat specification is reused. Thus the statement

```
FORMAT(F7.2,2(E15.5,E15.4),I7)
```

causes the format

```
F7.2,2(E15.5,E15.4),I7
```

to be used on the first record and the format

```
2(E15.5,E15.4),I7
```

on succeeding records.

As a further example, consider

```
FORMAT(F7.2/(2(E15.5,E15.4),I7))
```

the first record has the format

```
F7.2
```

successive records have the format

```
2(E15.5,E15.4),I7
```

7.5.14 Formats Stored as Data

The character string comprising a format specification may be stored as the values of an array. Input-output statements may reference the format by giving the array name rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT". The enclosing parentheses are included.

As an example, consider the sequence

```
      DIMENSION SKELETON(2)
      READ (5,1), (SKELETON(I),I = 1,2)
    1 FORMAT (2A8)
      READ (5,SKELETON) K,X
```

The first READ statement enters the character string into the array SKELETON. In the second READ statement, SKELETON is referenced as the format governing conversion of K and X.

7.5.15 Carriage Control for Printing

Every record that is transmitted to a listing device for printing is assumed to have a carriage control character as the first character of the record. The carriage control character itself is not printed. The carriage control characters are:

Character	Function Before Printing
blank	Space one line
0	Space two lines
1	Skip to first line of next page
+	No space (for overprinting)

Any other character is treated as a blank.

EXAMPLE:

```
10 FORMAT(9H1   PAGE ,I3/1H0)
```

7.6 Auxiliary I/O Statements

These statements are used to control the positioning and file marking of sequential files.

7.6.1 REWIND Statement

FORM:

```
REWIND u
```

where u is an I/O unit designation.

This statement directs the I/O unit designated to reposition to the first record. U may not be an asynchronous variable.

EXAMPLES:

```
REWIND 2
REWIND K
```

7.6.2 BACKSPACE Statement

FORM:

```
BACKSPACE u
```

where u is an I/O unit designation.

This statement directs the I/O unit designated to backspace one record. U may not be an asynchronous variable.

EXAMPLES:

```
BACKSPACE 5
BACKSPACE N
```

7.6.3 END FILE Statement

FORM:

```
END FILE u
```

where u is an I/O designation.

The statement directs the I/O unit designated to write an end file mark. U may not be an asynchronous variable.

EXAMPLES:

```
END FILE 4
END FILE T
```

8. PROGRAM UNITS

A FORTRAN program consists of one main program and, optionally, SUBROUTINE subprograms, FUNCTION subprograms, and BLOCK DATA subprograms. Each of these is termed a "program unit."

8.1 PROGRAM Statement

FORM:

```
PROGRAM identifier
```

The PROGRAM statement defines the program name that is used as the entry point name for the object module. The identifier may not appear anywhere else in the program unit. This statement, if present, must be the first statement of a main program, if not present the main program name defaults to F%MAIN.

8.2 END Statement

FORM:

```
END
```

The END statement must be the physically last statement of each program unit. It informs the compiler of the end of the program unit. The END statement must be on a single source line; continuation lines are not allowed.

8.3 RETURN Statement

FORM:

```
RETURN
```

This statement returns control from a FUNCTION or SUBROUTINE subprogram to the calling program unit. Normally, the last statement executed in a subprogram is a RETURN statement. It need not be physically the last statement of the program. Any number of RETURN statements may be used.

8.4 RESUME Statement

FORM:

RESUME

This statement is used to place a function or subroutine in the asynchronous or parallel mode. It allows the calling program unit to continue execution; however, the function or subroutine continues to execute as well until a RETURN statement is executed.

8.5 Subprogram Communications

The main program and subprograms communicate with each other by means of COMMON variables and parameters. If the means of communication is by parameters, the arguments of the subroutine or function call are known as actual parameters. Corresponding arguments in the subroutine or function argument list are formal parameters.

8.5.1 Actual Parameters

The actual parameters which appear in a subroutine call or a function reference may be any of the following:

An arithmetic expression
A logical expression
A constant
A simple variable
An array element reference
An array name
A FUNCTION name
A SUBROUTINE name

8.5.2 Formal Parameters

The formal parameters appearing in the parenthetical list of a FUNCTION or SUBROUTINE statement may be any of the following:

An array name
A simple variable
A subprogram name
(either function or subroutine)

The formal parameters are replaced at each execution of the subprogram by the actual parameters supplied in the CALL statement or function reference.

Formal parameters representing array names must appear within the subprogram in type or DIMENSION statements giving dimension information. In a type or DIMENSION statement, formal parameters may be used to specify variable dimensions for array name formal parameters. Variable dimensions may be given only for arrays which are formal parameters.

Within a subprogram, the use of formal parameters is restricted as follows:

1. Formal parameters may not appear in COMMON statements.
2. Formal parameters may not appear in EQUIVALENCE statements.
3. Formal parameters may not appear in DATA statements.

8.5.3 Correspondence Between Actual and Formal Parameters

When a subprogram is called, the formal parameters must agree with the actual parameters as to number, order, type, and length. For example, if an actual parameter is a integer constant than the corresponding formal parameter must be of INTEGER type.

Also, the formal and actual parameters must be either both synchronous or both asynchronous, they may not be mixed.

If a formal parameter is an array name, the corresponding actual parameter may be either an array name or an array element.

If a formal parameter is assigned a value in the subprogram, the corresponding actual parameter must be a simple variable, array element, or array name. A constant or expression should not be used as an actual parameter if the corresponding formal parameter may be assigned a value.

8.6 Statement Function Definition Statement

FORM:

```
identifier(identifier,identifier,...)=expression
```

This statement defines an internal subprogram. The entire definition is contained in a single statement. The first identifier is the name of the subprogram being defined.

Statement function subprograms are functions; they are single-valued and must have at least one argument. The type of the function is determined by the type of the function identifier.

The identifiers enclosed in parentheses represent the arguments of the function. These are formal parameters and have meaning and must be unique only within the statement. They may be identical to identifiers of the same type appearing elsewhere in the program. These identifiers must agree in order, number, type, and length with the actual parameters given at execution time.

The use of a parameter in the defining expression is specified by the use of its parameter identifier. Expressions are the only permissible arguments of internal functions; hence the

parameter identifiers may appear only as simple variables in the defining expression. They may not appear as array identifiers.

Identifiers appearing in the defining expression which do not represent parameters are treated as ordinary variables.

The defining expression may include references to external functions or other previously defined internal functions.

All statement function definition statements must precede the first executable statement of the program.

EXAMPLES:

```
SSQR(K) = K*(K+1)*(2K+1)/6
NOR(T,S)= .NOT.(T.OR.S)
ACOSH(X)= (EXP(X/A)+EXP(-X/A))/2
```

In the last example above, X is a parameter identifier and A is an ordinary identifier. At execution the function is evaluated using the current value of the quantity represented by A.

8.7 FUNCTION Subprograms

A FUNCTION subprogram is a function; it returns a single value and is referenced as a basic element in an expression. A FUNCTION subprogram begins with a FUNCTION declaration and return control to the calling program by means of a RETURN or RESUME statement. It is a program unit and, consequently, must terminate with an END statement.

8.7.1 FUNCTION Statement

FORM:

```
FUNCTION identifier(identifier,identifier,...)
```

This statement declares the program which follows to be a function subprogram. The first identifier is the name of the function being defined. This identifier must appear as a simple variable and be assigned a value during execution of the subprogram. This value is the function value.

Identifiers appearing in the list enclosed in parentheses are formal parameters representing the function arguments.

EXAMPLE:

```
FUNCTION FLOAT (I)
FLOAT = I
RETURN
END
```

8.7.2 FUNCTION Type

The type of the function is the type of identifier used to name the function. This identifier may be typed, implicitly or explicitly, in the same way as any other identifier. Alternately, the function may be explicitly typed in the FUNCTION statement itself by replacing the word FUNCTION with one of the following:

```
INTEGER FUNCTION
REAL FUNCTION
DOUBLE FUNCTION
DOUBLE PRECISION FUNCTION
COMPLEX FUNCTION
LOGICAL FUNCTION
```

For example, the statement

```
COMPLEX FUNCTION HPRIME(S,N)
```

is equivalent to the statements

```
FUNCTION HPRIME(S,N)
COMPLEX HPRIME
```

EXAMPLES:

```
FUNCTION MAY(RANG,XP,YP,ZP)
REAL FUNCTION COT(ARG)
```

8.7.3 Library Functions

The FORTRAN system supplies a library of standard functions which may be referenced from any program. Appendix A lists these library functions. These are divided into two sets: basic external functions and intrinsic functions. The basic external functions are called by the object program in the same manner as normal, user-supplied functions. Intrinsic function names are known to the compiler and intrinsic function references may be treated in non-standard ways (such as as expanding the function in-line). The programmer can supply his own function in place of an intrinsic function by including the name in EXTERNAL statements in all calling programs.

8.8 SUBROUTINE Subprograms

A SUBROUTINE subprogram is not a function; it ~~may be multi-valued and~~ can be referred to only by a CALL or CREATE statement. A SUBROUTINE subprogram begins with a SUBROUTINE declaration and returns control to the calling program by means of a RETURN or RESUME statement.

8.8.1 SUBROUTINE Statement

FORMS:

```
SUBROUTINE identifier
SUBROUTINE identifier(identifier,identifier,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. The identifiers in the list enclosed in parentheses are formal parameters.

A SUBROUTINE subprogram may use one or more of its formal parameters to represent results. The subprogram name is not used for return of results.

A SUBROUTINE subprogram need not have any parameters at all.

EXAMPLES:

```
SUBROUTINE EXIT
SUBROUTINE FACTOR(CEF,N,ROOTS)
SUBROUTINE RESIDUE (NUM,D,DEN,M,RES)
```

8.8.2 CALL Statement

FORMS:

```
CALL identifier
CALL identifier(argument,argument,...,argument)
```

The CALL statement is used to transfer control (call) to a subroutine subprogram. The identifier is the subprogram name.

The parameters may be expressions, array identifiers, alphanumeric strings, or subprogram identifiers, as in the case of a function reference. Unlike a function, however, a subroutine ~~may have more than one value and~~ cannot be referenced as a basic element in an expression. A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used.

EXAMPLES:

```
CALL EXIT
CALL SWITCH(SIN,2.LE.BETA,X**4,Y)
CALL MULT(A,B,C)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments.

8.8.3 CREATE Statement

FORMS:

```
CREATE identifier
CREATE identifier(argument,argument,...,argument)
```

The CREATE statement is used to execute a subroutine as a parallel process. The identifier is the subroutine name.

The CREATE statement is the only way to initiate execution of any asynchronous subroutine.

EXAMPLES:

```
CREATE    GRAPH($IN,X)
CREATE    PROC
```

APPENDIX A: LIBRARY FUNCTIONS

Intrinsic Functions

Function	Type of Parameter	Type of Result	Definition
ABS(a)	Real	Real	$\|a\|$
IABS(a)	Integer	Integer	
DABS(a)	Double	Double	
AINT(a)	Real	Real	Truncation
INT(a)	Real	Integer	
IDINT(a)	Double	Integer	
AMOD(a_1,a_2)	Real	Real	$a_1 \pmod{a_2}$
MOD(a_1,a_2)	Integer	Integer	
AMAXO(a_1,a_2,...)	Integer	Real	Max(a_1,a_2,...)
AMAX1(a_1,a_2,...)	Real	Real	
MAXO (a_1,a_2,...)	Integer	Integer	
MAX1 (a_1,a_2,...)	Real	Integer	
DMAX1(a_1,a_2,...)	Double	Double	
AMINO(a_1,a_2,...)	Integer	Real	Min(a_1,a_2,...)
AMIN1(a_1,a_2,...)	Real	Real	
MINO (a_1,a_2,...)	Integer	Integer	
MIN1 (a_1,a_2,...)	Real	Integer	
DMIN1(a_1,a_2,...)	Double	Double	
FLOAT(a)	Integer	Real	Conversion from integer to Real

FULL(a)	any asynchronous type	logical	Test Full semaphore
EMPTY(a)	any asynchronous type	logical	Test Empty semaphore
IFIX(a)	Real	Integer	Conversion from real to integer
SIGN(a_1,a_2)	Real	Real	Sign of a_2 times $\|a_1\|$
ISIGN(a_1,a_2)	Integer	Integer	
DSIGN(a_1,a_2)	Double	Double	
DIM (a_1,a_2)	Real	Real	a_1-Min(a_1,a_2)
IDIM (a_1,a_2)	Integer	Integer	
SNGL(a)	Double	Real	Coversion from double to real
REAL(a)	Complex	Real	Obtain real part of complex
AIMAG(a)	Complex	Real	Obtain imaginary part of complex
DBLE(a)	Real	Double	Conversion from real to double
CMPLX(a_1,a_2)	Real	Complex	$a_1 + a_2\sqrt{-1}$
CONJG(a)	Complex	Complex	Obtain conjugate of complex

IOR (a_1,a_2)	Integer	Integer	Inclusive OR
LAND(a_1,a_2)	Integer	Integer	Logical AND
NOT (a_1)	Integer	Integer	Logical negation
IEOR(a_1,a_2)	Integer	Integer	Exclusive OR
ISHFT(a_1,a_2)	Integer	Integer	Shift a_1 by a_2 bits

BASIC EXTERNAL FUNCTIONS

Function	Type of Parameter	Type of Result	Definition
EXP(a)	Real	Real	e^a
DEXP(a)	Double	Double	
CEXP(a)	Complex	Complex	
ALOG(a)	Real	Real	ln (a)
DLOG(a)	Double	Double	
CLOG(a)	Complex	Complex	
ALOG10(a)	Real	Real	$\log_{10}(a)$
DLOG10(a)	Double	Double	
SIN(a)	Real	Real	sin (a)
DSIN(a)	Double	Double	
CSIN(a)	Complex	Complex	
COS(a)	Real	Real	cos (a)
DCOS(a)	Double	Double	
CCOS(a)	Complex	Complex	
TANH(a)	Real	Real	tanh (a)
SQRT(a)	Real	Real	$(a)^{\frac{1}{2}}$
DSQRT(a)	Double	Double	
CSQRT(a)	Complex	Complex	
ATAN(a)	Real	Real	arctan (a)
DATAN(a)	Double	Double	
ATAN2 (a_1, a_2)	Real	Real	arctan (a_1/a_2)
DATAN2(a_1, a_2)	Double	Double	
DMOD(a_1, a_2)	Double	Double	$a_1 \pmod{a_2}$
CABS(a)	Complex	Complex	Absolute value of complex

APPENDIX B: COMPILER LISTING FORMAT AND DIAGNOSTICS

Compiler Listing Format

The compiler optionally outputs the following listings:

- Source and diagnostic
- Allocation
- Cross Reference
- Object
- Called subprograms
- Statement label
- Line number listing
- Summary

The source and diagnostic listing includes a sequential line number, source image and interspersed diagnositcs.

The allocation listing includes the name, mode, size and location of each variable. Allocation errors are listed.

The cross reference listing includes all variable names in alphabetical order, their location and the line number of each reference. The line number is followed by an asterisk if the reference is a possible modification. References to all labels are included.

The object listing is produced only for the verification of object code by the developers and maintainers. It includes the location, symbolic opcode and operands of each instruction.

The called subprogram listing includes the name, mode and number of parameters of each called subprogram.

The statement label listing includes the label, location and type of use of all statement labels.

The line number lisitng includes each line number and its location.

The summary listing includes the number of errors, the number of warnings, the program size and the data size.

Statement Error Diagnostics

During compilation, statements which violate the syntactic or semantic rules of the language are recognized and error indications are printed.

There are two levels of statement diagnostics: warnings and errors. Warnings are issued for minor infractions where the compiler can still determine what is to be done and compile the statement. Errors are severe violations of the language. In the case of errors, compilation proceeds as if the statement was never encountered. The statement label, if any, remains defined. If the error statement is ever executed, it will cause a link to a system routine which will terminate execution of the program and notify the user that an attempt has been made to execute an erroneous statement. The name of the program and the line number of the statement will be displayed.

One character of the statement is marked with an up-arrow symbol "^" output directly beneath the erroneous character, for example:

```
ZATA = X + Y * - A
                ^
```

The character "-" is marked as an error.

In the case of a snytax error, the marked character itself was unacceptable, as in the example above. In the case of a semantic error, an identifier or other construct is in error, the mark indicating the last character of the construct. For example, in the line:

```
COMMON ALPHA, BETA, ALPHA, GAMMA
       ^
```

the mark indicates that the identifier ALPHA is misused.

The compiler attempts all interpretations of statement type before discarding a statement. The marked position indicates the greatest amount of correct information found under the most logical assumption of statement type.

A comment specifying the reason for the failure is output directly after the marked line. There may be more than one diagnostic per line. The diagnostics are listed left-to-right. Each diagnostic is followed by a sequence of characters: "E*E*E....E" or "W*W*W....W" indicating "error" or "warning", respectively.

An alphabetic list of possible statement diagnostics follows:

ARGUMENT CONVERTED (Warning)

The type of the indicated parameter for an intrinsic function was converted to agree with the type required by the function.

ARGUMENT COUNT (Warning)

The number of parameters to a subprogram is wrong either because it is an intrinsic function which the compiler knows about or because the same subprogram was called previously with a different number of parameters.

ARRAY SIZE

An attempt has been made to declare an array that has more elements than allowed.

BLOCK DATA ONLY

A DATA statement not in a BLOCK DATA subprogram attempted to initialize a variable in COMMON.

An executable statement has been included in a BLOCK DATA subprogram.

CONSTANT SIZE

The size of the indicated constant is outside the allowable range.

DATA TYPE

The type of a constant in a DATA statement does not agree with the type of the variables it is to initialize.

DATA COUNT

The number of variables in a DATA statement does not agree with the number of constants.

DECLARATION CONFLICT

An attempt has been made to declare an identifier as a FORTRAN entity (simple variable, array, subprogram, statement function name) which has already been used otherwise.

DIMENSION RANGE

An array dimension has been declared outside of the allowable range.

DUPLICATE DUMMY

A formal parameter has been declared twice in a statement function definition, FUNCTION, or SUBROUTINE statement.

EXTRA COMMA (Warning)

More than one comma has been encountered at a point where a single comma was expected.

FORMAT LABEL

The indicated statement number was declared in the label field of a FORMAT statement and is being used in some manner other than as a format reference.

ILLEGAL DO CLOSE (Warning)

A DO loop was closed with an illegal statement.

ILLEGAL LABEL

A statement number is less than 1 or greater than 99999. Or, a DO statement references a previously defined label or a label previously referenced as a FORMAT.

ILLEGAL NUMBER

FORMAT, DATA or CONST repeat count not greater than zero. Or, uninary minus of Hollerith or Hexadecimal constant. Or, illegal complex number format.

JUMP LABEL

A statement number which is not a FORMAT label has been used as if it were. Or, a FORMAT label has been previously referenced by an IF or a GOTO.

LABEL MISSING (Warning)

The indicated statement cannot be executed because it has no statement number. Or, the indicated FORMAT cannot be used because it has no statement number.

MISSING COMMA (Warning)

A comma was missing at a point where one was expected but compilation could continue.

MISUSED NAME

An identifier has been used in the wrong context, such as:

- A formal parameter in a DATA or EQUIVALENCE statement.
- A variable dimension which is not a simple formal parameter.
- A subprogram name used without parameters in an expression.

MULTI DEFINED

A statement number is multiply defined.

NOT ARRAY

An identifier which is not an array name has been used where an array name should have appeared.

NOT INTEGER

A variable or expression of type other than integer has been used where only integer type is allowed.

NUMBER OF SUBSCRIPTS

The number of subscripts in an array reference is incorrect.

RANGE

The second character in the declaration of an IMPLICIT range does not alphabetically follow the first character.

A constant subscript array reference has a subscript which falls outside the size of the array.

STATEMENT NOT ALLOWED

A statement has been used in an illegal context. An illegal logical IF secondary statement or the statement is in the wrong order, such as a statement function defintion not preceding executable statements.

SYNTAX

Usually erroneous punctuation or an illegally constructed expression.

The character marked shows how much of the statement was scanned before it ceased to make sense.

TYPE CONFLICT

The same first character has been declared different types in IMPLICIT.

The types of the operands of an arithmetic or logical operator are illegal.

The types of the right and left-hand sides of an assignment are improper.

UNDIMENSIONED

A simple variable is followed by a left parenthesis.

UNRECOGNIZABLE

The entire statement was unrecognizable.

UNSUCCESSFUL COPY

A copy statement could not be performed.

Program Error Diagnostics

After the source program has been listed, summary error messages pertaining to the program as a whole are listed.

The following describes each of these messages or set of messages.

FUNCTION NAME NOT REFERENCED

This message appears at the end of any FUNCTION subprogram wherein the function name does not appear on the left hand side of an assignment statement.

OPEN DO LOOPS

Following this heading, all DO loops which were not closed are listed in the form:

"statement-number OPENED AT LINE line-number"

UNDEFINED LABELS

All undefined statement numbers are listed after this heading. Each undefined statement number appears as:

statement-number FIRST REFERENCED AT LINE line-number

ALLOCATION ERRORS

This heading will be followed by a list of identifiers that were incorrectly assigned memory locations by the program. These errors are caused by COMMON and/or EQUIVALENCE statements. Such errors as:

- Equivalencing variables from different common blocks.
- Extending a common block backwards.
- Specifying an impossible equivalence group.

APPENDIX C: COMPILER EXECUTION

The method by which the compiler is envoked is not known at present, however the various compilation options are.

Compiler Options

Conditional Compilation - COND

This option causes lines containing a "D" in column one to be compiled, default is to be treated as a comment.

Debug Mode - DEBUG

This option causes the compiler to generate additional code to produce a block level trace at execution time.

Cross Reference - XREF

This option causes the compiler to print an alphabetical listing of each identifier and where it was defined, altered, or referenced.

List Object - LOBJ

This option causes the compiler to print pseudo assembly language statements corresponding to the object code generated.

APPENDIX F: FORTRAN RUNTIME LIBRARY

The programs involved with the process of executing a FORTRAN program may be divided into four catagories:

Function Library
Arithmetic Library
Resident I/O Library
Non-Resident I/O Library

The Function Library consists of the basic external functions and non-inline intrinsic functions. These are all referenced directly by the FORTRAN program. Hence, the module name is the same as the FORTRAN function name. The initial version of the function library will be coded in FORTRAN for expediency.

The Function Library is comprised of the following modules:

ALOG	CLOG	DCOS	EXP
ALOG10	COS	DEXP	IDINT
ATAN	COSH	DLOG	SIN
ATAN2	CSIN	DLOG10	SINH
CABS	CSQRT	DMOD	SQRT
CCOS	DATAN	DSIN	TANH
CEXP	DATAN2	DSQRT	

The Arithmetic Library consists of the functions indirectly referenced by the FORTRAN Program. These modules all have the prefix FR% to identify them as part of the FORTRAN runtime library and to limit conflicts with user names. Like the Function Library, the majority of the Arithmetic Library is written in FORTRAN.

Name	Language	Function
FR%CVDI	Assembly	Convert double precision to integer.
FR%CVID	Assembly	Convert integer to double precisio
FR%DADD	Assembly	Double precision add
FR%DSUB	Assembly	Double precision subtract
FR%DMUL	Assembly	Double precision multiply
FR%DDIV	Assembly	Double precision divide
FR%CMUL	Fortran	Complex multiply
FR%CDIV	Fortran	Complex divide
FR%DTNE	Fortran	Double precision test .NE.
FR%DTEQ	Fortran	Double precision test .EQ.
FR%DTLE	Fortran	Double precision test .LE.
FR%DTLT	Fortran	Double precision test .LT.
FR%DTGE	Fortran	Double precision test .GE.
FR%DTGT	Fortran	Double precision test .GT.
FR%CTNE	Fortran	Complex test .NE.
FR%CTEQ	Fortran	Complex test .EQ.
FR%PWII	Fortran	I**I power routine
FR%PWIR	Fortran	I**R power routine
FR%PWID	Fortran	I**D power routine
FR%PWIC	Fortran	I**C power routine
FR%PWRI	Fortran	R**I power routine

Name	Language	Function
FR%PWRR	Fortran	R**R power routine
FR%PWRD	Fortran	R**D power routine
FR%PWRC	Fortran	R**C power routine
FR%PWDI	Fortran	D**I power routine
FR%PWDR	Fortran	D**R power routine
FR%PWDD	Fortran	D**D power routine
FR%PWDC	Fortran	D**C power routine
FR%PWCI	Fortran	C**I power routine
FR%PWCR	Fortran	C**R power routine
FR%PWCD	Fortran	C**D power routine
FR%PWCC	Fortran	C**C power routine

The Resident I/O Library consists of the modules required to initiate FORTRAN I/O and communicate with the Non-Resident Library located, presently, on the Interdata computer. These modules all reside in the HEP, are written in assembler and have the prefix FR%.

Name	Function
FR%ENF	Initiate ENCODE
FR%DNF	Initiate DECODE
FR%WNF	Initiate WRITE formatted
FR%WRF	Initiate WRITE formatted (with ERR=option)
FR%WNU	Initiate WRITE unformatted
FR%WRU	Initiate WRITE unformatted (with ERR=option)
FR%RNF	Initiate READ formatted
FR%RDF	Initiate READ formatted (with END=option)
FR%RRF	Initiate READ formatted (with ERR=option)
FR%RBF	Initiate READ formatted (with END=,ERR=options)
FR%RNU	Initiate READ unformatted
FR%RDU	Initiate READ unformatted (with END=option)

Name	Function
FR%RRU	Initiate READ unformatted (with ERR=option)
FR%RBU	Initiate READ unformatted (with END=,ERR=options)
FR%BKSP	Initiate BACKSPACE
FR%ENDF	Initiate ENDFILE
FR%REWD	Initiate REWIND
FR%STOP	Initiate STOP
FR%PAUS	Initiate PAUSE
FR%IIOF	Formatted I/O, Integer item
FR%RIOF	Formatted I/O, Real item
FR%DIOF	Formatted I/O, Double item
FR%CIOF	Formatted I/O, Complex item
FR%LIOF	Formatted I/O, Logical item
FR%SIOF	Formatted I/O, Stop I/O
FR%IIOU	Unformatted I/O, Integer item
FR%RIOU	Unformatted I/O, Real item
FR%DIOU	Unformatted I/O, Double item
FR%CIOU	Unformatted I/O, Complex item
FR%LIOU	Unformatted I/O, Logical item
FR%SIOU	Unformatted I/O, Stop I/O
FR%TENT	Trace option, subprogram entry
FR%TRTN	Trace option, RETURN statement
FR%TRES	Trace option, RESUME statement
FR%TFLW	Trace option, trace flow
FR%CERR	Report compilation error
FR%RERR	Report runtime error
~~FRUNERR~~	Function Library interface to FR%RERR

The Non-Resident I/O Library consists of the modules required to perform the logical and physical I/O on the Interdata computer. These modules all reside on the Interdata, are written in assembler and have the prefix FR$.

Name	Function
FR$PIO	Perform physical I/O
FR$IOF	Perform formatted logical I/O
FR$IOU	Perform unformatted logical I/O
FR$RIO	Interpret D,E,F and G formats
FR$IIO	Interpret Z,I and L formats
FR$FUT	Formatted I/O utility modules