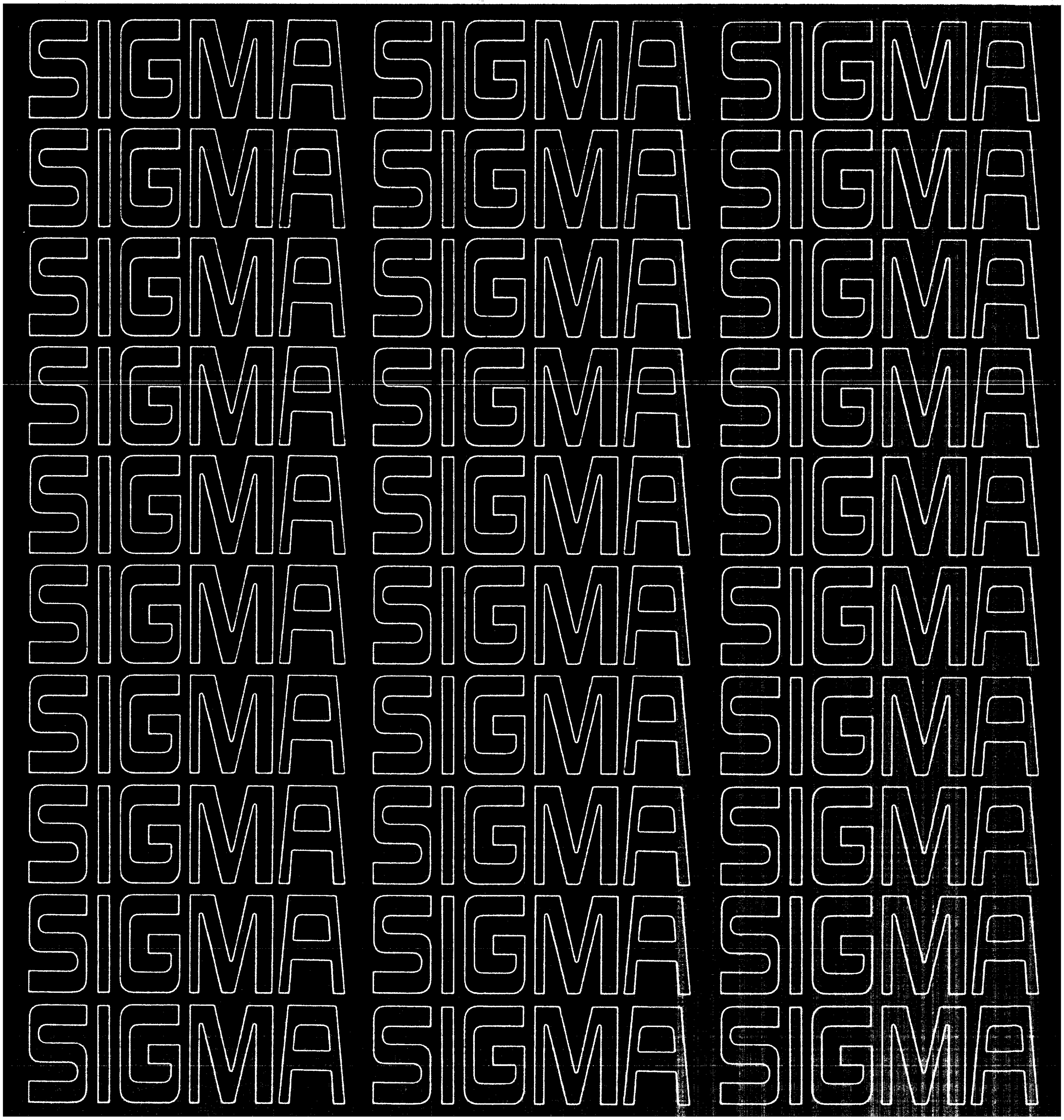
SDS SIGMA 5/7 FORTRAN IV-H

Reference Manual

SCIENTIFIC DATA SYSTEMS



Price: $3.00

FORTRAN IV-H REFERENCE MANUAL

for

SDS SIGMA 5/7 COMPUTERS

90 09 66C

August 1968

SDS

SCIENTIFIC DATA SYSTEMS/701 South Aviation Boulevard/El Segundo, California 90245

REVISION

This publication, SDS 90 09 66C, is a minor revision of FORTRAN IV-H Reference Manual, 90 09 66B (dated January 1968). A change in text from that of the previous manual is indicated by a vertical line in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
Sigma 7 Computer Reference Manual	90 09 50
Sigma 5 Computer Reference Manual	90 09 59
Sigma 5/7 Basic Control Monitor Reference Manual	90 09 53
Sigma 5/7 Symbol/Meta-Symbol Reference Manual	90 09 52
Sigma 7 Mathematical Routines Technical Manual	90 09 06
Sigma 5/7 FORTRAN IV-H Operations Manual	90 11 44
Sigma 5/7 FORTRAN IV-H Library/Run-Time Technical Manual	90 11 38

CONTENTS

APPENDIXES

ILLUSTRATIONS

TABLES

INTRODUCTION

SDS Sigma FORTRAN IV-H is a one-pass compiler that operates under the Basic Control Monitor (BCM). It is designed for maximum compatibility with both ASA Standard FORTRAN and IBM 360 H-level FORTRAN IV.

SDS Sigma FORTRAN IV-H includes a number of features not found in ASA FORTRAN. Among these features are:

- ENTRY statement
- Double complex data
- FORTRAN II READ, PRINT, and PUNCH statements
- IMPLICIT statement
- END and ERROR options on READ statements
- T (tab) format
- NAMELIST input/output
- Object program listing

Additionally, the facility of introducing in-line assembly language coding into FORTRAN programs is an available option, where compatibility with other FORTRAN systems is not a factor.

The compiler tables, such as symbol and label tables, are dynamically allocated by FORTRAN IV-H to optimize memory usage.

1. FORTRAN IV-H PROGRAMS

SDS FORTRAN IV-H programs are comprised of an ordered set of statements that describe the procedure to be followed during execution of the program and the data to be processed by the program. Some data values to be processed may be external to the program and read into the computer during program execution. Similarly, data values generated by the program can be written out while processing continues. Statements belong to one of two general classes:

1. executable statements[t], that perform computation, input/output operations, and program flow control.
2. nonexecutable statements[t], that provide information to the compiler about storage assignments, data types and program form, as well as providing information to the program during execution about input/output formats and data initialization.

Statements defining an SDS FORTRAN IV-H program follow a prescribed format. Figure 1 is a sample FORTRAN Coding Form. Each line on the form consists of 80 spaces or columns; however, the last eight columns are used only for identification or sequence numbers and have no effect on the program. Columns 1 through 72 are used for the statements.

The first field, columns 1 through 5, is used for statement labels. Statement labels allow statements to be referenced by other portions of the program. Labels are written as decimal integers, with all blanks (leading, embedded, or trailing) ignored. Section 5, "Control Statements", contains a more extensive discussion of statement labels.

The body of each statement is written in columns 7 through 72, but if additional space is required, a statement may be continued. FORTRAN IV-H accepts an unlimited number of continuation lines. Each continuation line must contain a character other than blank or zero in column 6. The initial line of each statement contains only the characters blank or zero in column 6. If a statement is labeled, the label must appear on the initial line of the statement; labels appearing on continuation lines are ignored.

Column 1 may contain the character C to indicate that the line is to be treated as a comment only, with no effect upon the program. Comment lines may appear anywhere in the program, except within a statement (i.e., interspersed with continuation lines).

Statements may have blanks inserted as desired to improve readability, except within literal fields (e.g., in Hollerith constants and in FORMAT statements).

The set of characters acceptable to SDS Sigma FORTRAN IV-H is

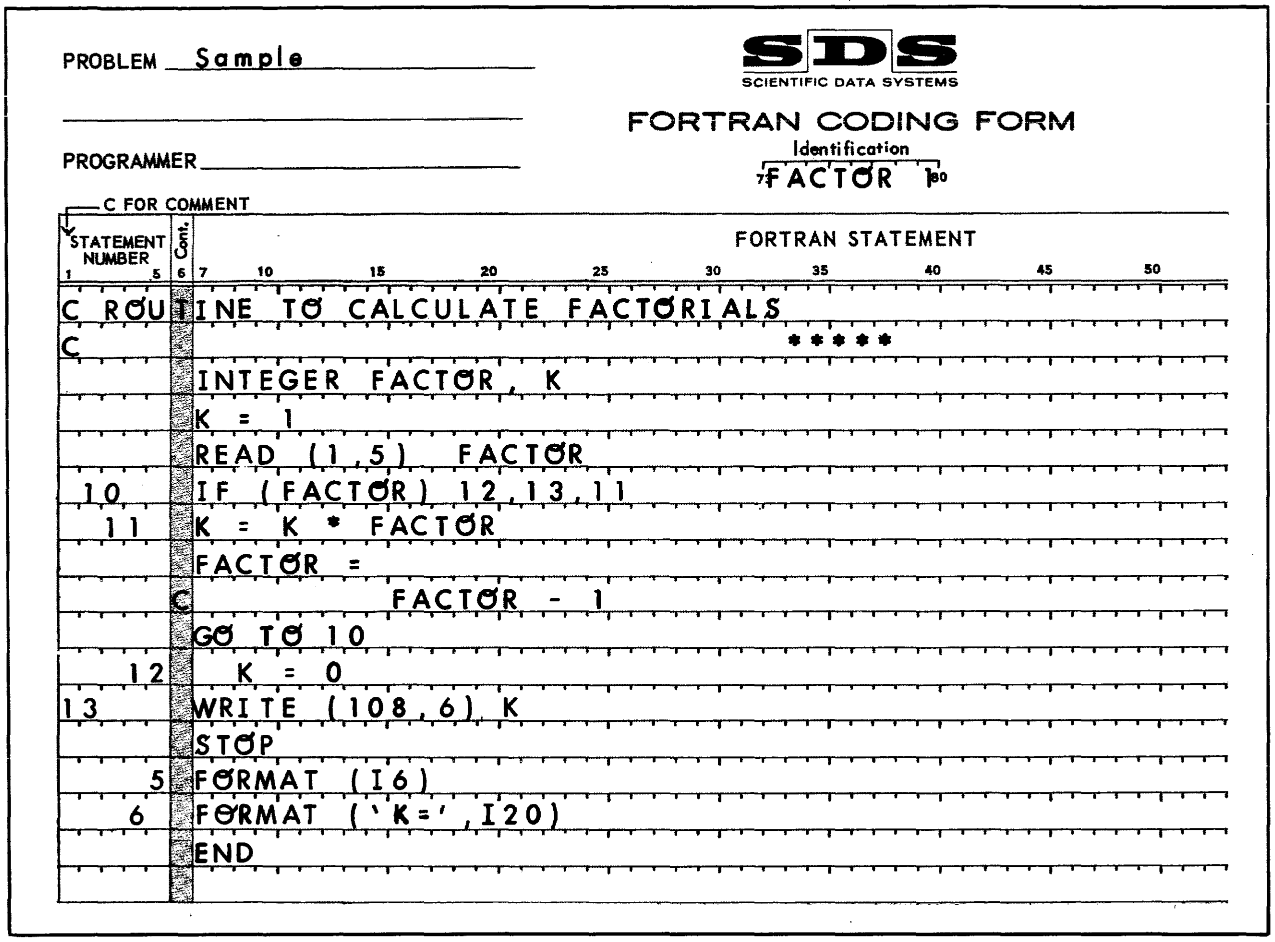
Letters[tt]:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Special characters: (useful)[tt]	+ - * / = ( ) . , $ ' & blank
Special characters: (other)	< > ; : @ # % ¢ ? ! – | _ "

This character set conforms to the Extended Binary-Coded Decimal Interchange Code (EBCDIC) standard. (See Appendix A.)

Figure 1 illustrates a sample FORTRAN IV-H program. An explanation is given in Table 1.

[t] See Appendix B.

[tt] The dollar sign ($) character is accepted, though not recommended, as a letter of the alphabet. It may therefore be used in FORTRAN identifiers, such as $, FIVE$, or $300. For the purposes of the IMPLICIT statement (see Chapter 7), $ follows Z in the set of letters.



PROBLEM Sample

PROGRAMMER

SDS SCIENTIFIC DATA SYSTEMS

FORTRAN CODING FORM

Identification FACTØR

```
C RØUTINE TØ CALCULATE FACTØRIALS
C                                            *****
      INTEGER FACTØR, K
      K = 1
      READ (1,5) FACTØR
   10 IF (FACTØR) 12,13,11
   11 K = K * FACTØR
      FACTØR =
     C          FACTØR - 1
      GØ TØ 10
   12   K = 0
13    WRITE (108,6) K
      STØP
    5 FØRMAT (I6)
    6 FØRMAT ('K=',I20)
      END
```

Figure 1. Sample SDS FORTRAN IV-H Program

Table 1. Sample Program

Line	Meaning
1,2	The character C in column 1 defines these lines as comments.
3	A nonexecutable statement that defines to the compiler the variables FACTOR and K as integers.
4	An assignment statement that sets K equal to 1.
5	An input command that causes the value of FACTOR to be read into storage. The value is read from unit 1. The form in which the value of FACTOR appears external to the computer is specified by FORMAT 5 (line 14).
6	Statement 10 tests the value of FACTOR and transfers control to statement 11, 12, or 13 as follows: If FACTOR < O, control is transferred to statement 12. If FACTOR = O, control is transferred to statement 13. If FACTOR > O, control is transferred to statement 11.
7	Statement 11 is another assignment statement that assigns to K the value of the expression K times factor. In other words, the current value of K is replaced by the current value of K multiplied by the value of FACTOR.

Table 1. Sample Program (cont.)

Line	Meaning
8	The statement appearing on lines 8 and 9 is an assignment statement, written as an initial line and one continuation line.
9	The C in column 6 causes line 9 to be a continuation of line 8. This statement assigns to FACTOR the value of the current value of FACTOR minus 1.
10	When the GO TO statement is executed, an unconditional transfer of control to statement 10 (line 6) occurs.
11	Statement 12, an assignment statement, assigns the value zero to the variable K.
12	The WRITE output statement, 13, causes the name of the variable K and its value to be written out on unit 108, which is normally assigned to the Printer (see statement 6, line 15 for designated FORMAT statement).
13	The control statement STOP causes execution of the program to be terminated.
14	FORMAT statement corresponding to READ statement on line 5.
15	FORMAT statement corresponding to WRITE statement on line 12.
16	The END line informs the processor during compilation that it has reached the physical end of the source program.
	In this program, if the value of FACTOR is initially 3 as read by line 5, statement 10 will be executed four times, the statements on line 7 through 10 will be executed three times, and the statements on lines 4, 5, 12, and 13 will be executed once each.

2. DATA

Numerical quantities – constants and variables – as distinguished in FORTRAN IV-H are a means of identifying the nature of the numerical values encountered in a program. A constant is a quantity whose value is explicitly stated. For example, the integer 5 is represented as "5"; the number π, to three decimal places, as "3. 142". A variable is a numerical quantity that is referenced by name rather than by its explicit appearance in a program statement. During the execution of the program, a variable may take on many values rather than being restricted to one. A variable is identified and referenced by an identifier.

All data processed by an SDS FORTRAN IV-H program can be classed as one of seven types:

Integer	Double Complex
Real	Logical
Double-Precision	Literal
Complex	

LIMITS ON VALUES OF QUANTITIES

Integer data are precise representations of the range of integers from -2, 147, 483, 648 to +2, 147, 483, 647; that is, -2^{31} to $+2^{31} - 1$. Integer data may only be assigned integral values within this range.

Real data (sometimes known as floating-point data) can be assigned approximations of real numbers, the magnitudes of which are within the range 5.398×10^{-79} to 7.237×10^{75} (i.e., 16^{-65} to 16^{63}). A real datum may acquire positive or negative values within this range or the value zero. Real data have an associated precision of 6+ significant digits. That is, the sixth most significant digit will be accurate, while the seventh will sometimes be accurate, depending on the value assigned to the datum.

Double-precision-data may approximate the identical set of values as real data. However, double-precision data have an associated precision of 15+ significant digits.

Complex data are approximations of complex numbers. These approximations take the form of an ordered pair of real data. The first of the two real data approximates the real part, and the second real datum approximates the imaginary part of the complex number. The values that may be assigned to each part are identical to the set of values for real data.

Double complex data have the same form as complex data except that both the real and imaginary parts are double-precision values.

Logical data can acquire only the values "true" or "false".

Literal Data are character strings of up to 255 characters. Like logical data, literal data do not have numeric values. Any of the characters discussed in Section 1 may appear in literal data.

CONSTANTS

Constants are data that do not vary in value and are referenced by naming their values. There are constants for each type of data. Although numeric constants are considered as being unsigned, they may be preceded by the plus or minus operators. The operator is not considered part of the constant, however. (See Section 3.)

Integer Constants

Integer constants are represented by strings of digits. The magnitude of an integer constant must not exceed 2, 147, 483, 647.

Examples:

```
382    997263    1000000000    000546    8
 13      1961     323344224    382437    0
```

Real Constants

Real constants are represented by strings of digits with a decimal point and/or an exponent. The exponent follows the numeric value and consists of the letter E, followed by a signed or unsigned 1- or 2-digit integer that represents the power of ten by which the numeric value is to be multiplied. Thus, the following forms are permissible:

```
n.m       n.      .m
n.mE±e    n.E±e   nE±e
```

where

n, m, and e are strings of digits.

The plus sign preceding e is optional.

For example, .567E5 has the meaning $.567 \times 10^5$ and can also be represented by any of the following equivalent forms:

```
0.567E+05     5.67E4    56700.
567000.E-1    567E02    56700.000E-00
```

The value of a real constant may not exceed the limits for real data. Any number of digits may be written in a real constant, but only the 7 most significant digits are retained.

Since any real constant may be written in a variety of ways, the user has freedom of choice regarding form.

Examples:

```
5.0     7.6E+5     3.14159265358979323846
0.01    6.62E-37   .58785504
```

Double-Precision Constants

Double-precision constants are formed exactly like real constants, except that the letter D is used instead of E in the exponent. To denote a constant specifically as double-precision, the exponent must be present. Thus, a double-precision constant may be written in any of the following forms:

```
n.mD±e    n.D±e    nD±e
```

where

n, m, and e are strings of digits

D signifies a double-precision constant

The plus sign preceding e is optional.

The value of a double-precision constant may not exceed the limits for double-precision data. Any number of digits may be written in a double-precision constant, but only the 15 most significant digits are retained.

Examples:

```
1.2345678765432D1    576.3D+01    312.D-4
.9963D+3             .1254D-02    885.D+3
```

Complex Constants

Complex constants are expressed as an ordered pair of constants in the form

(c_1, c_2)

where

c_1 and c_2 are signed or unsigned, real constants.

The complex constant (c_1, c_2) is interpreted as meaning $c_1 + c_2 i$, where $i = \sqrt{-1}$. Thus, the following complex constants have values as indicated:

(1.34,52.01)	=	1.34 + 52.01i
(98.344E11,34452E-3)	=	983.44 + 34.452i
(-1.,-1000.)	=	-1.0 - 1000.0i

Neither part of a complex constant may exceed the value limits established for real data.

Double Complex Constants

Double complex constants are formed in exactly the same way as complex constants. If either the real or imaginary part is a double-precision constant, the complex constant becomes a double complex constant.

Examples:

(.757D6,3D-4)	=	757000.0D0 + .0003D0i
(7.,0D0)	=	7.0D0 + 0.0D0i
(-4.286D0,1.3)	=	-4.286D0 + 1.3D0i

Neither part of a double complex constant may exceed the value limits established for double-precision data.

Logical Constants

Logical constants may assume either of the two forms

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

where these forms have the logical values "true" and "false", respectively.

Literal Constants

A literal constant has the form

```
's'
```

where

s is a string of up to 255 alphanumeric and/or special characters. Note that blanks are significant in such character strings.

Within a literal constant, two consecutive quotation marks may be used to represent a single quotation mark (or apostrophe). For example, 'AB''CD' represents the five characters AB'CD. However, quotation (') marks separated by blanks are not considered to be consecutive.

Examples:

```
'ALPHANUMERIC INFORMATION'
'''DON''T!'''
```

Literal constants can appear in three contexts:

1. An argument to a function or subroutine
2. A constant item in a DATA statement
3. A PAUSE or STOP statement ('s' form only)

A literal constant cannot appear as an element of an expression.

IDENTIFIERS

Identifiers are strings of letters and decimal digits, the first of which must be a letter,[t] used to name variables as well as subprograms and COMMON blocks. (See Chapters 7 and 8 for discussions of COMMON and subprograms.)

Identifiers in SDS FORTRAN IV-H may consist of up to six alphanumeric characters. Blank characters embedded in identifiers are ignored; therefore, ON TIME and ONTIME are identical. There are no restricted identifiers in SDS FORTRAN IV-H; however, for clarity, it is advisable not to use identifiers that correspond to SDS FORTRAN IV-H statement types.

Examples:

```
X      A345Q     STRESS     J3     MELVIN     QUANTY
ELEVAT     I     L9876     DIFFER     SETUP
```

VARIABLES

Variables are data whose values may vary during program execution and are referenced with an identifier. Variables may be any of the data types. (There is no such entity as a literal variable; any type of variable may contain a literal string. Normally, integer variables are used.)

If a variable has not been assigned to a particular data type (see "Classification of Identifiers", Chapter 7), the following implicit typing conventions are assumed:

1. Variables whose identifiers begin with the letters I, J, K, L, M, or N are classified as integer variables.
2. Variables whose identifiers begin with any other letter are classified as real variables.

These classifications are referred to as the "IJKLMN rule".

Consequently, double-precision, complex, double-complex, and logical variables must be explicitly declared as such (see "Explicit Type Statements" in Chapter 7). The values assigned to variables may not exceed the limits established for the applicable data types.

SCALARS

A scalar variable is a single datum entity accessed via an identifier of the appropriate type.

Examples:

```
J1     NAME     SCALAR    EQUATE     E     NEW     DHO     XXX8
```

ARRAYS

An array is data in which the data form an ordered set. Associated with an array is the property of dimension. SDS FORTRAN IV-H arrays may have up to seven dimensions and are referenced by an identifier. For a complete discussion on arrays see "Array Declarations" in Chapter 7.

Array Elements

An array element is a member of the set of data comprising an array. Array elements are referenced by the array identifier, followed by a list of subscripts enclosed in parentheses

$v(s_1, s_2, \ldots, s_n)$

where:

v	is the array name
s_i	is a subscript (see below)
n	is the number of subscripts, which must be equal to the number of dimensions of the array ($0 < n \leq 7$)

Subscripts

A subscript can be any expression that has a resultant mode of integer.

[t]See Chapter 1.

The evaluated result for a subscript must always be greater than zero. For example, if an array element is designated as ALPHA(K-4), the value of K must be greater than 4.

Examples:

Arrays	Subscripts	Array Elements
MATRIX	(3, 9, 5, 7, 6, 1, 2)	MATRIX(3, 9, 5, 7, 6, 1, 2)
CUBE	(5*J, P, 3)	CUBE(5*J, P, 3)
DATA	(I, J, K, L, M, N)	DATA(I, J, K, L, M, N)
J35Z	(I+4, 6*KRAN-2, ITEMP)	J35Z(I+4, 6*KRAN-2, ITEMP)
BOB	(3, IDINT(DSQRT(D)))	BOB(3, IDINT(DSQRT(D)))

FUNCTIONS

Functions are subprograms that are referenced as basic elements in expressions. A function acts upon one or more quantities, called arguments, and produces a single quantity, called the function value. The appearance of a function reference constitutes a reference to the value produced by the function, when operating on the given argument. A function reference is denoted by the identifier that names the function, followed by a list of arguments enclosed in parentheses

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

where

f	is the name of the function
a_i	is an argument. Arguments may be constants, variables, expressions, or array or subprogram names (see "Arguments and Dummies", Chapter 8).

Functions are classified in the same way as variables; that is, unless the type is specifically declared otherwise, the IJKLMN rule applies.[t] The type of a function is not affected by the type of its arguments.

Examples of function references are:

SIN(A+B)   CHECK(7.3, J, ABS(Y))   KOST(ITEM)

Many library functions are provided in SDS FORTRAN IV-H. In addition, the user may define his own functions (see Chapter 8).

[t]For certain library functions, this does not hold true; see "Basic External Functions" in Chapter 8.

3. EXPRESSIONS

Expressions are strings of operands separated by operators. Operands may be constants, variables, or function references. An expression may contain subexpressions; subexpressions are expressions enclosed in parentheses. Operators may be unary — that is, they may operate on a single operand. They may also be binary, operating on pairs of operands. Expressions may be classed as arithmetic, logical, or relational. All expressions are single valued, and the evaluation of any expression has a unique result.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of integer, real, double-precision, complex, and/or double-complex constant, variable, or function references connected by arithmetic operators.

The arithmetic operators are:

Operator	Operation
+	Addition (binary) or Positive (unary)
-	Subtraction (binary) or Negative (unary)
*	Multiplication
/	Division
**	Exponentiation

Arithmetic expressions may be of a relatively simple form

```
A
-TERM
1.2607
ACE - DEUCE
W9OML * DE + W9CMI / XKA9RU
F(5.8E2) - A / B9J(L)
```

or the more complicated form

```
X + (I12 * (G) ** L(3) + N / SDS) - (H)
-B + SQRT(B ** 2 - 4 * A * C) + T * (S + B / I * (K(J) / (V1 - V0) + (Z1 - Z0)))
(X + Y) ** 3 + 0.7352986E-7
-((M + N) * (Z - Q(J)))
```

Evaluation Hierarchy

The expression

```
A + B / C
```

might be evaluated as

```
(A + B) / C
```

or as

```
A + (B / C)
```

Actually, the latter form is the way the expression is interpreted without explicit grouping. This example illustrates that it is necessary to formulate rules for expression evaluation so that such ambiguities do not occur.

Subexpressions have been defined as expressions enclosed in parentheses. It is also possible to have nested subexpressions as in

```
X * (Z + Y * (H - G / (I + L) - W) + M(8))
```

where (I + L) may be called the innermost subexpression, and (H - G / (I + L) - W) is the next innermost subexpression. The evaluation hierachy is, therefore, as follows:

1. The innermost subexpression, followed by the next innermost subexpression, until all subexpressions have been evaluated.

2. The arithmetic operations, in the following order of precedence:

Operation	Operator	Order
Exponentiation	**	1 (highest)
Multiplication and Division	* /	2
Addition and Subtraction	+ -	3

Some additional conventions are necessary.

1. At any one level of evaluation, operations of the same order of precedence (except for exponentiation) are evaluated from left to right. Consequently, I / J / K / L is equivalent to ((I / J) / K) / L.

2. Consecutive exponentations are performed right to left, thus

   ```
   A ** B ** C
   ```

   is interpreted as

   ```
   A ** (B ** C)
   ```

   The use of parentheses is recommended, as many FORTRAN systems interpret consecutive exponentiation differently.

3. The sequence "operator operator" is not permissible. Therefore, A * -B must be expressed as A * (-B).

4. As an algebraic notation, parentheses are used to define evaluation sequences explicitly. Thus, $\frac{A + B}{C}$ is written as (A + B) / C.

   Example:

   The expression

   ```
   A * (B + C * (D - E / (F + G) - H) + P(3))
   ```

   is evaluated in the sequence

   $r_1 = F + G$

   $r_2 = E / r_1$

   $r_3 = D - r_2 - H$

   $r_4 = C * r_3$

   $r_5 = B + r_4 + P(3)$

   $r_6 = A * r_5$

   where the r_i are the various levels of evaluation.

MIXED EXPRESSIONS

When an arithmetic expression contains elements of more than one type, it is known as a mixed expression. Logical elements may not appear in an arithmetic expression except as function arguments (see rule 2, below). When an expression contains more than one type of element, the mode of the expression is determined by the type and length specifications of its elements. Table 2 illustrates how the mode for mixed expressions is determined.

Table 2. Mode of Mixed Expressions Using Operators + - * /

+ - * /	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	DOUBLE COMPLEX
INTEGER	Integer	Real	Double Precision	Complex	Double Complex
REAL	Real	Real	Double Precision	Complex	Double Complex
DOUBLE PRECISION	Double Precision	Double Precision	Double Precision	Double Complex	Double Complex
COMPLEX	Complex	Complex	Double Complex	Complex	Double Complex
DOUBLE COMPLEX	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

It can be seen that a hierarchy of type and length specifications exists. The order of precedence is:

Type	Precedence
Double Complex	1 (highest)
Complex or Double Precision	2
Real	3
Integer	4

Within a mixed expression, elements of lower precedence type are converted to the higher type before being combined with other elements. For example, (3/4) is an integer expression and has the value zero, while ((3/4)+0.0) is a real expression and has the value 0.75. Parentheses do not affect the mode of computation.

The following rules also apply to mixed expressions:

1. Subscripts and arguments are independent of the expression in which they appear. These expressions are evaluated in their own mode (i.e., integer) and neither affect nor are affected by the mode of the outer expression.

2. Only expression elements of the types shown in Table 3 may be combined with an exponentiation operator.

Table 3. Valid Type Combinations for Exponentiations

Base		Exponent
Integer Real Double Precision	**	Integer Real Double Precision
Complex Double Complex	**	Integer

The mode of the results of an exponentiation operation can be determined in the same manner as that for other arithmetic operations (see Table 1).

4. Complex and double-precision elements have the same level of precedence. If an expression contains both of these types, it acquires double-complex type. This is the only case in which an expression may have a type that is higher than (or different from) all its constituents.

5. Integer, real, and double-precision values that appear in complex or double-complex expressions are assumed to have imaginary parts of zero.

6. Values of expressions, subexpressions, and elements may not exceed the value limits associated with the mode of the expression.

RELATIONAL EXPRESSIONS

The form of a relational expression is

$e_1 \; r \; e_2$

where

e_1 and e_2	are arithmetic expressions whose mode is integer, real, or double-precision
r	is a relational operator (see below)

Evaluations of relational expressions result in either of the two values "true" or "false", i.e., relational expressions are of logical type.

Relational operators cause comparisons between arithmetic expressions.

Operator	Meaning
.LT.	Less than (<)
.LE.	Less than or equal to (≤)
.EQ.	Equal to (=)
.NE.	Not equal to (≠)
.GE.	Greater than or equal to (≥)
.GT.	Greater than (>)

Examples:

1. LT. 6	is true.
0. GT. 8	is false.
0. LT. (2. ** N)	is always true, while
0. LT. - (2. ** N)	is always false.

When two arithmetic expressions are compared, using a relational operator, the two expressions are first evaluated, each in its own mode. The comparison is then made in the mode of higher precedence; i.e., the value of the lower mode expression is converted to the mode of higher precedence.

A test for equality between real or double-precision quantities may not be meaningful on a binary machine. Since these quantities are only approximations to most values, numbers that are "essentially" equal may differ by a small amount in their binary representations. It can only be said that computations whose operands and results have exact binary representations will produce these results.

It is not permissible to nest relational expressions such as

```
(L.LT. (X .GT. 0.2345E6))
```

where (X .GT. 0.2345E6) is a relational subexpression, rather than an arithmetic expression, as the definition of relational expressions requires.

LOGICAL EXPRESSIONS

Logical expressions are expressions of the form

$e_1\ c_1\ e_2\ c_2\ e_3\ c_3 \ldots\ e_n$

where

e_i	are logical elements.
c_i	are the binary logical operators (see below).

Evaluations of logical expressions result in either of the two values, "true" or "false".

Logical elements are defined as one of the following entities:

1. a logical variable or logical function reference
2. a logical constant
3. a relational expression
4. any of the above enclosed in parentheses
5. a logical expression enclosed in parentheses
6. any of the above, preceded by the unary logical operator .NOT.

Logical Operators

There are three logical operators:

Operator	Type
.NOT.	unary
.AND.	binary
.OR.	binary

Table 4 illustrates the meanings of the logical operators.

1. .NOT. e is "true" only when e is "false".
2. e_1 .AND. e_2 is "true" only when both e_1 and e_2 are "true".
3. e_1 .OR. e_2 is "true" when either or both e_1 and e_2 are "true".

Table 4. Evaluation of Logical Expressions

Expression Values		Evaluation		
		.NOT. e	e_1 .AND. e_2	e_1 .OR. e_2
e True	——	False	——	——
e False	——	True	——	——
e_1 False	e_2 False	——	False	False
e_1 True	e_2 False	——	False	True
e_1 False	E_2 True	——	False	True
e_1 True	e_2 True	——	True	True

Evaluation Hierarchy

Parentheses are used to define evaluation sequences explicitly, in a manner similar to that discussed for arithmetic expressions. Consequently,

A .AND. B .OR. Q(3) .NE. X

does not have the same meaning as

A .AND. (B .OR. Q(3) .NE. X)

where (B .OR. Q(3) .NE. X) may be called a logical subexpression.

The evaluation hierarchy for logical expressions is

1. arithmetic expressions
2. relational expressions (the relational operators are all of equal precedence).
3. the innermost logical subexpression, followed by the next innermost logical subexpression, etc.
4. the logical operations, in the following order of precedence:

Operator	Order
.NOT.	1 (highest)
.AND.	2
.OR.	3

For example, the expression

L .OR. .NOT. M .AND. X .GE. Y

is interpreted as

L .OR. ((.NOT. M) .AND. (X .GE. Y))

Note: It is permissible to have two contiguous logical operators only when the second operator is .NOT.; in other words

e_1 .AND. .OR. e_2

is not valid, while

e_1 .AND. .NOT. e_2

is legal. Two consecutive .NOT. operators are not permissible. The logical expression to which the operator .NOT. applies should be enclosed in parentheses if it contains two or more quantities. For example, if X and Z are logical variables having the values TRUE and FALSE, respectively, the following expressions are not equivalent:

.NOT. X .AND. Z

.NOT. (X .AND. Z)

In the first expression .NOT. X is evaluated first and produces the value FALSE. This, when ANDed with Z (also, FALSE), results in the value FALSE for the expression.

In the second expression X .AND. Z is evaluated first and produces the value FALSE. Then the value FALSE is NOTed, resulting in the value TRUE for the expression.

4. ASSIGNMENT STATEMENT

Many kinds of statements are included in the SDS FORTRAN IV-H language. The most basic of these is the assignment statement, which defines a computation to be performed and is used in a manner similar to equations in normal mathematical notation.

A simple assignment statement has the form

v = e

where

v	is a variable (a scalar or an array element of any type)
e	is an arithmetic or logical expression. (v must be a logical variable only if e is a logical expression)

This statement means, "assign to v the value of the expression e." It is not an equation in the true sense; it does not declare that v <u>is</u> equal to e, but rather it <u>sets</u> v equal to e. Thus, the statement

N = N + 1

is not a contradiction: it increments the current value of N by 1.

The expression need not be the same type as the variable, although in practice it usually is. When it is not, the expression is evaluated in its own mode, independent of the type of the variable. Then, if permissible, it is converted to the type of the variable according to Table 5 and assigned to the variable.

Table 5. Mixed Variable Types and Expression Modes

Variable Type	Expression Mode					
	integer	real	double precision	complex	double complex	logical
integer	X	I	I	I	I	N
real	F	X	P	R	R	N
double precision	F	P	X	D	D	N
complex	R	R	R	X	P	N
double complex	D	D	D	P	X	N
logical	N	N	N	N	N	X

The symbols used in Table 5 have the following meanings:

<u>Symbol</u>	<u>Meaning</u>
X	Direct assignment of the exact value.
I	The value is truncated to integer. The truncated value is equal to the sign of the expression times the greatest integer less than or equal to the absolute value of the expression (e.g., 4274.9983 is truncated to 4274, and -0.6 to 0). Values that are greater than the maximum size of an integer will be truncated at the high-order end as well. Results in this case generally are not meaningful.
F	The variable is assigned the real or double-precision approximation of the value. Since real precision is less than that of integers, conversion to real precision may cause a loss of significant digits.

Symbol	Meaning
P	The precision of the value is increased or decreased accordingly.
R	The real part of the variable is assigned the real approximation of the expression. The imaginary part of the variable is set to zero.
D	The real part of the variable is assigned the double-precision approximation of the expression. The imaginary part of the variable is set to zero.
N	Not allowed.

Examples:

```
A = B
Q(I) = Z ** 2 + N * (L - J)
L = F .OR. .NOT. C .AND. (R . GE. 23. 9238E-1)
CRE(8, ED) = R (ALL, MEN)
PI = 4 * (ATAN(0. 5) + ATAN(0. 2) + ATAN(0. 125))
```

5. CONTROL STATEMENTS

Each executable statement in a FORTRAN IV-H program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement.

LABELS

If program control is to be transferred to a particular statement, that statement must be identified. Statements are identified by labels. Nonexecutable statements may have labels, but, except for FORMATs, the labels should not be referenced.

Statement labels consist of up to five decimal digits and must be greater than zero. Embedded blanks and leading zeros are not significant. The following labels are equivalent.

```
857    00857    8 5 7    085 7
```

Statement labels may be assigned in any order; their numerical values have no effect on the sequence of statement compilation or execution.

GO TO Statements

GO TO statements transfer control from one point in a program to another. FORTRAN IV-H includes three forms of GO TO statements: unconditional, assigned, and computed.

Unconditional GO TO Statement

This statement has the form

```
GO TO k
```

where k is a statement label. The result of the execution of this statement is that the next statement executed is the one whose label is k. For example, in

```
        :
        GO TO 502
98      X = Y
        :
502     A = B
        :
```

statement 502 will be executed immediately after the GO TO statement.

Assigned GO TO Statement

The format of the assigned GO TO is

GO TO v,[$(k_1, k_2, k_3, \ldots, k_n)$]

where

v is a nonsubscripted integer variable that has been assigned (via an ASSIGN statement, see below) one of the statement labels $k_1 - k_n$.

k_i is a statement label (the list enclosed in brackets is optional).

Each label appearing in the list must be defined in the program in which the GO TO statement appears (i.e., must be the label of a program statement). This statement causes control to be transferred to the statement label (k_i) that corresponds to the current assignment of the variable (v).

Examples:

```
ASSIGN 5371 TO LOC
.
.
.
GO TO LOC,(117,56,101,5371)
```

The GO TO statement transfers control to the statement labeled 5371. Note that v (the variable "LOC" in the above example) must have been set by a previously executed ASSIGN statement prior to its execution in the GO TO statement.

Computed GO TO Statement

The computed GO TO statement is expressed as

GO TO ($k_1,k_2,k_3,\ldots,k_n$), v

where

k_i	is a statement label
v	is a nonsubscripted integer variable whose value determines to which of the k_i control will be transferred.

This statement causes control to be transferred to the statement whose label is k_j, where j is the integer value of the variable v, for $1 \le j \le n$. If j is not between 1 and n, no transfer occurs, and control passes to the statement following the computed GO TO statement. In most previous FORTRAN systems, this situation has been considered an error, but is no longer so considered.

Examples:

Statement	Expression Value	Transfer to
GO TO (98,12,405,3), N	3	405
GO TO (1,8,7,562), I	2	8
GO TO (4,88,1), N	0	next statement
GO TO (63,9,3,2), J	8	next statement

ASSIGN Statement

The ASSIGN statement, used to assign a label to a variable, has the form

ASSIGN k TO v

where

k	is a statement label
v	is a nonsubscripted integer variable

Examples:

```
ASSIGN 5 TO JUMP
ASSIGN 22 TO M
ASSIGN 1234 TO IRETURN
ASSIGN 99999 TO IERROR
```

A variable that has had a label assigned to it may be used only in an assigned GO TO statement.

A variable that has most recently had a label assigned to it should not be used as a numeric quantity. Conversely, a variable that has not been assigned a label may not appear in any context requiring a label. The following case illustrates improper usage:

```
ASSIGN 703 TO HI
A = HI / LOW
```

This usage is not permissible because the value of HI is indeterminate, since its value depends on where the program is loaded. Furthermore,

```
M = 5
```

cannot be substituted for

```
ASSIGN 5 to M
```

or vice versa, because the integer constant "5" is implied in the first case, and the label "5" in the second.

IF Statements

Very often it is desirable to change the logical flow of a program on the basis of some test. IF statements, which may be called conditional transfer statements, are used for this purpose. There are two forms of IF statements: arithmetic and logical.

Arithmetic IF Statement

The format for arithmetic IF statements is

IF (e) k_1, k_2, k_3

where

e is an expression of integer, real, or double-precision modes.

k_1, k_2, and k_3 are statement labels.

The arithmetic IF statement is interpreted to mean

IF $e < 0$, GO TO k_1

IF $e = 0$, GO TO k_2

IF $e > 0$, GO TO k_3

If e is a real or double-precision expression, a test for exact zero may not be meaningful on a binary machine. If the expression involves any amount of computation, a very small number is more likely to result than an exact zero. For this reason, floating point arithmetic IF statements generally should not be programmed to have a unique branch for zero.

Examples:

Statement	Expression value	Transfer to
IF (K) 1,2,3	47802	3
IF (3 * M(J) - 7) 76,4,3	-6	76
IF (C(J, 10) + A / 4) 23, 12, 12	0.0002	12
IF (NEXT + LAST) 3, 156, 3	0	156

Logical IF Statement

The logical IF statement is represented as

IF (e) s

where:

e is a logical mode expression

s is any executable statement except a DO statement or another logical IF statement

The statement s is executed if the expression e has the value "true"; otherwise, the next executable statement following the logical IF statement is executed. The statement following the logical IF will be executed in any case after the statement s, unless the statement s causes a transfer.

Examples:

```
IF (FLAG .OR. L) GO TO 3135

IF (OCTT * TRR .LT. 5.334E4) CALL THERMAL

IF (.NOT. SWITCH2) REWIND 3
```

CALL Statement

This statement, used to call or transfer control to a subroutine subprogram (see Chapter 8), may take either of the following forms:

CALL p

CALL p $(a_1, a_2, a_3, \ldots, a_n)$

where

p is the identifier of the subroutine subprogram.

a_i is an argument, which may be any of the following: Constants, subscripted or nonsubscripted variables, arithmetic expressions, statement label arguments ($\&a_i$, where a_i is the statement label), or array or subprogram names. (See "Arguments and Dummies", Chapter 8.)

A subroutine is similar to a function except that it does not necessarily return a value, and must not, therefore, be used in an expression. Furthermore, while a function must have at least one argument, a subroutine may have none. For example,

```
CALL CHECK
```

Arguments that are scalars, array elements, or arrays may be modified by a subroutine, effectively returning as many results as desired. The following call might be used to invert the matrix A, consisting of K rows and columns, store the resulting matrix in the array B, and set D(J) equal to the determinate of B.

```
CALL INVERT(A, K, B, D(J))
```

A complete discussion of the usage and forms of arguments to supprograms is contained in Chapter 8.

A subroutine name has no type (e.g., real, integer) associated with it; it merely identifies the block of instructions to be executed as a result of the CALL. Therefore, the appearance of a subprogram name in a CALL statement does not cause it to take on any implicit type.

Other examples of CALL statements are given below. Statement labels are identified by a preceding ampersand.

```
CALL ENTER(&44, N)

CALL RX23A(X ** Y - 7,0, SQRT(A * A + B * B) / DIV, TEST)

CALL EVALUE
```

RETURN Statement

The RETURN statement causes an exit from a subprogram. It takes one of the forms

```
RETURN
RETURN  v
```

where v is an integer constant or INTEGER variable whose value must be greater than zero, but no greater than the number of asterisks that appear in the SUBROUTINE statement (see "SUBROUTINE Subprograms" and "Arguments and Dummies" in Chapter 8 for a discussion on the use of asterisks in SUBROUTINE statements).

A RETURN statement must be, chronologically, the last statement executed in any subprogram, but it need not be last physically. There may be any number of RETURN statements in a subprogram. A RETURN statement in a main program will be treated as a STOP statement.

The first form, RETURN (without the v) is the statement usually used. In a subroutine, it returns control from the subroutine to the first executable statement following the CALL statement that called the subroutine. In a function, it causes the latest value assigned to the function name to be returned, as the function value, to the expression in which the function reference appeared. (See also, "FUNCTION Subprograms", Chapter 8.)

The second form, RETURN v, is used to provide an alternate exit from a SUBROUTINE subprogram. The value of v is used to determine which statement label in the calling argument list will be used as the return. The vth asterisk (counting from left to right in the SUBROUTINE statement) corresponds to the statement label that will be used. If the most recent entry to the subprogram did not contain any asterisks in the dummy list, the RETURN statement will cause a run-time diagnostic to be produced.

Examples:

Calling Program	Subprograms
33 CALL IT (LOCK, RET, QR, &11, &883)	SUBROUTINE IT (i, X, P, *, *)
⋮	⋮
66 X(8) = Y(C, K) + CHEBY(Z, Y)	RETURN 1
⋮	⋮
	RETURN 2
	END
	FUNCTION CHEBY (ARG, EXP)
	⋮
	RETURN
	END

When subroutine IT is called by statement 33, return is to statement 11 if the RETURN 1 exit is executed, or to statement 883 if the RETURN 2 exit is executed. When the function subprogram CHEBY is called by statement 66, the return from the function is to the point of call in 66.

DO Statement

These statements are used to control the repetitive execution of a group of statements. The number of repetitions depends on the value of a variable. The DO statement may be written

DO k v = e_1, e_2, e_3, or

DO k v = e_1, e_2

where

k	is a statement label not defined before the DO statement.
v	is a nonsubscripted integer variable.
e_1, e_2, and e_3	are integer constants greater than zero or unsigned nonsubscripted integer variables whose value is positive.

In the second form, e_3 and the preceding comma are omitted; in this case the value 1 is assumed for e_3.

A DO statement indicates that the block of statements following it are to be executed repetitively. Such a block is called a DO loop, and all statements within it, except for the opening DO statement, constitute the range of the DO statement. The last statement in a DO loop is the terminus and bears the statement label k.

The execution of a DO loop proceeds in the following manner:

1. The variable v is assigned the value of e_1.
2. The range of statements is executed for one iteration.
3. After each iteration, the value of v is incremented by the value of e_3. If e_3 is not present, the value 1 is used.
4. The value of v is then compared with the terminal value (e_2).
5. If v is greater than e_2, control is passed to the statement following the terminus (i.e., to the statement following the one whose label is k). Otherwise, the process is repeated from step 2.
6. The actual number of iterations defined by the DO statement is given by

$$\max\left(\left[\frac{e_2 - e_1}{e_3}\right] + 1, 1\right) \text{ for } e_3 \neq 0$$

   where the brackets represent the largest integral value not exceeding the value of the expression.

The range of a DO loop will always be executed at least once, even if the conditions for termination are met initially. For this reason, it is recommended that initially satisfied DO loops should not be used, especially since other FORTRAN systems may interpret this situation differently.

The terminal statement of a DO range (i.e., the statement whose label is k) may be any executable statement other than one of the following:

DO statement	RETURN statement
GO TO statement	STOP statement
Arithmetic IF statement	PAUSE statement

Logical IF statements are specifically allowed as terminal statements of a DO range.

Example:

```
22      DO 54 I = 1,15
25      SUM = SUM + Q(I)
        IF (SUM .LT. 0.0) SUM = 0.0
        SIGMA = SUM + R(I)
        IF (SIGMA - H ** 3 / T) 54,54,12
54      CONTINUE
12      L = Y(I)
```

In the example that begins with statement 22, the range of statements 25 through 54 will be executed 15 times, unless the arithmetic IF statement causes a transfer to statement 12. If all 15 iterations are completed, control is passed to statement 12 at the end of the fifteenth iteration.

If the range of a DO loop terminates on a statement with 'S' in column 1 (in-line coding), note that the code generated for the end of a DO loop immediately follows that statement.

Example:

```
        DO 10 I = 1,10
        X = X + I
S10     LW,3 X
        < end DO loop code >
S       STW,3 Z
```

It is not a recommended practice to put a DO close on an in-line coded statement. This is because the compiler can not diagnose incorrect terminal statements of a DO range, such as

```
          DO  10 I = 1, 10
          X = Y
    S10   B         20S
          .
          .
    20    Y = X
```

The value of the variable v appearing in a DO statement depends on the number of iterations completed. The value of v during any one iteration is

$e_1 = (i - 1) * e_3$

where i is the number of the current iteration, and e_1 and e_3 have the meanings discussed previously. If a transfer is made out of the range of a DO before all iterations have been completed, the value of v will be that of the iteration during which the transfer occurred. However, should the entire number of iterations be executed, the value of v is

$e_1 + m * e_3$

where m is the total number of iterations specified by the DO statement.

Thus, in the example beginning with statement 22, if all iterations are completed, statement 12 will be equivalent to

```
    12    L = Y(16)
```

However, if the arithmetic IF statement causes a transfer to statement 12 during the eighth iteration, the statement will mean

```
    12    L = Y(8)
```

The value of the indexing parameters (v, e_1, e_2, e_3) cannot be modified within the range of the DO, nor can they be modified by a subprogram called within the range of the DO.

A transfer out of the range of a DO loop is permissible at any time; however, a transfer into the range of a DO may only occur if there has been a prior transfer out of the DO range (assuming that none of the indexing parameters (v, e_1, e_2, e_3) are changed outside the range of the DO). For example:

```
          DO 25 H = K, Y, 1
          .
          .
          GO TO 8605
          .
          .
    24    A = H / 8
    25    JGU = Y(H) ** 3
          .
          .
    8605  R = SIN(G(H)) + JSU
          .
          .
    8606  GO TO 24
```

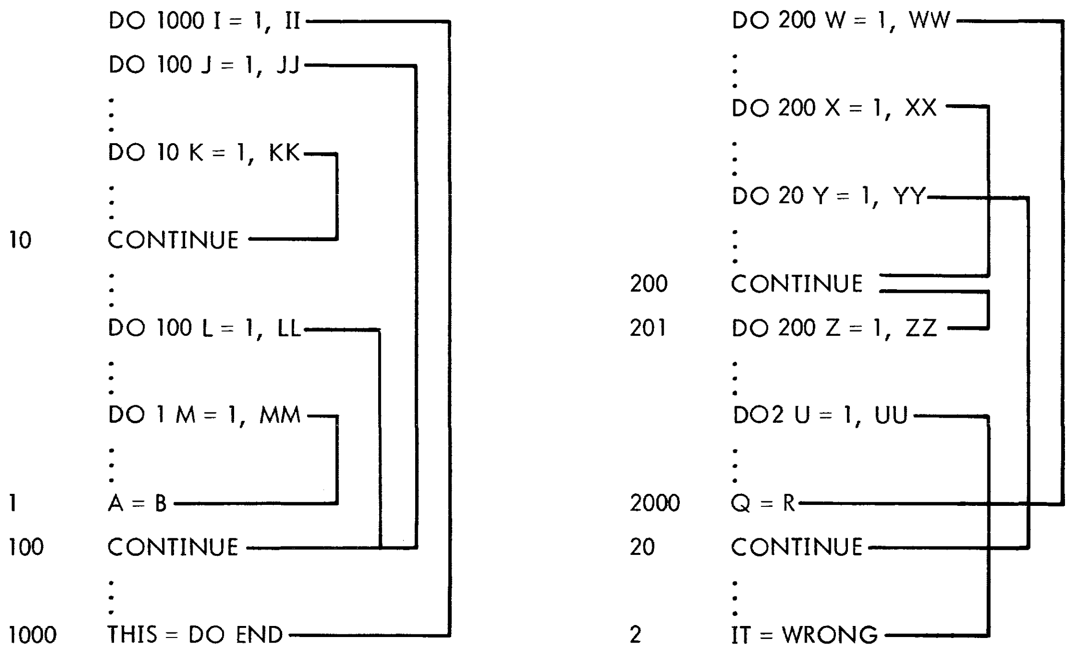
is permissible; in fact, the statements 8605 through 8606 are considered part of the DO range. The sequence

```
          GO TO 11
          .
          .
          DO 32 J= 2, 36, 2
    11    R(J) = 47. E-7 * T(J)
    32    T(J) = Q
          .
          .
```

is not valid because no transfer could possibly occur out of the DO range.

A DO loop may include another DO loop. Do loops may be nested; however, they cannot be overlapped. In a nest of DO loops, the same statement may be used as the terminal statement for any number of DO ranges; however, transfers to this statement can be made only from the innermost DO loop. There is no limit to the number of DO ranges that can be nested. Only if a transfer is made out of the range of the innermost DO loop can a return transfer into the range of nested DO loops be made. In this case, the return transfer must be to the innermost DO loop.

Examples:



Legal

```
        DO 1000 I = 1, II
        DO 100 J = 1, JJ
        :
        DO 10 K = 1, KK
        :
10      CONTINUE
        :
        DO 100 L = 1, LL
        :
        DO 1 M = 1, MM
        :
1       A = B
100     CONTINUE
        :
1000    THIS = DO END
```

Illegal

```
        DO 200 W = 1, WW
        :
        DO 200 X = 1, XX
        :
        DO 20 Y = 1, YY
        :
200     CONTINUE
201     DO 200 Z = 1, ZZ
        :
        DO2 U = 1, UU
        :
2000    Q = R
20      CONTINUE
        :
2       IT = WRONG
```

The terminal statement of a range may not physically precede the DO statement, as is shown in the case of statements 200 and 201 in the illegal example above.

CONTINUE Statement

This statement is written as

```
CONTINUE
```

and must appear in that form. The CONTINUE statement does not cause the compiler to generate machine instruction and, consequently, has no effect on a running program. The purpose of the CONTINUE statement is to allow the insertion of a label at any point in a program. For example:

```
      DO 72, I = 1, 20
      :
      IF (X ** I + 0.9999E-5) 72, 72, 88
72    CONTINUE
88    H(33) = T(3, R, L, E) / 22.5
      :
```

CONTINUE statements are most often used as the terminal statement of a DO range, as in the example above.

PAUSE Statement

PAUSE statements are written as

```
PAUSE
PAUSE n
PAUSE 's'
```

where

n is an unsigned integer constant of up to five digits ($1 \leq n \leq 5$).

's' is a literal constant.

This statement causes the program to cease execution temporarily, presumably for the purpose of allowing the computer operator to perform some specified action. The operator can then signal the program to continue execution, beginning with the statement immediately after the PAUSE.

If an integer or a literal constant is appended to the PAUSE statement, the word PAUSE and this value will be displayed to the computer operator when the program pauses; otherwise, the word PAUSE is displayed.

STOP Statement

STOP statements are written in the form

```
STOP
STOP n
```

where

n is an unsigned integer constant

This statement terminates the execution of a running program. If it appears within a subprogram, control is not returned to the calling program. If an integer is appended to the STOP statement, it will be output immediately before termination.

END Statement

An END statement is used to inform the FORTRAN IV-H compiler that it has reached the physical end of a program. The statement must appear in the form

```
END
```

If, in a main program, control reaches an END statement, the effect is that of a STOP statement. If, in a SUBROUTINE or FUNCTION subprogram, control reaches an END statement, the effect is that of a RETURN statement (see Chapter 8).

The following restriction applies to any statement that begins with the character string E N D:

> If the compiler has encountered only the characters END at the end of a FORTRAN IV-H line, it assumes that the statement is an END statement and will act accordingly. An END statement may not appear on a continuation line.

This limitation is due to an historic FORTRAN feature; namely, the way in which continuation is specified. As indicated by the following examples, certain statements, although legitimate FORTRAN IV-H statements, will be processed as though they were END statements.

```
          Processed as END Statements              Not Processed as END Statements

column: 6   7 . . . . . . . . . . . . . . . . .    6   7 . . . . . . . . . . . . . . . . .
            END                                        END FILE
        1   FILE 2                                 1   2
            END                                        END RA
        X             RATE = A * B                 X                      TE = A * B
            END                                        END (I, J
        X             (I, J, K) = .NOT. Q          X   , K) = .NOT. Q
                                                       E
                                                   1   N
                                                   2   D
```

Similarly, illegal statements of the same nature as those in the first column will be treated as END statements.

6. INPUT/OUTPUT

The FORTRAN language provides a series of statements that determine the control of and condition for data transmission between computer storage and external data handling devices, such as magnetic tape and paper tape handlers, typewriters, card units, and line printers. These statements are of three types:

1. READ and WRITE statements that cause specified lists of data to be transmitted between computer storage and one of the group of external devices
2. FORMAT statements used in conjunction with the input/output of formatted records to provide conversion and editing information that specifies their internal and external representation
3. Auxiliary I/O statements for positioning and demarcation of external files (as on magnetic tapes)

The data transmitted by input/output statements are transmitted as records of sequential information consisting of binary-coded strings of characters or unformatted binary values in a form similar to internal storage. For either type of transmission the I/O statements refer to external devices, lists of data names, and – for formatted data – to format specification statements.

INPUT/OUTPUT LISTS

An input/output list represents an ordered group of data names that identify the data to be transmitted and the order of their transmission. These lists have the form

$m_1, m_2, \ldots, m_n$

where

m_i are list items separated by commas, as shown.

LIST ITEMS

A list item may be either a single or multiple datum identifier.

A single datum identifier is the name of a scalar variable or an array element.

Examples:

A	B
MATRIX(25,L)	ALPHA(J,N)

Multiple data identifiers are in one of two forms:

1. An array name appearing in a list without subscripts is considered equivalent to the listing of each element in the array.

   Example:

   If B is a 2-dimensional array, the list item B is equivalent to

   B(1, 1), B(2, 1), B(3, 1), . . . , B(1, 2), B(2, 2), . . . , B(j, k)

   where

   j and k are the dimension limits of B

2. DO-implied items are lists of one or more identifers or other DO-implied items followed by a comma character and an expression of one of the forms

   $v = e_1, e_2, e_3$

   $v = e_1, e_2$

   enclosed in parentheses.

The elements v, e_1, e_2, and e_3 have the same meaning as defined for the DO statement. The items enclosed in parentheses with a DO implication are considered to be in the range of the DO implication. For input lists the indexing parameters v, e_1, e_2, and e_3 may appear in this range only as subscripts.

Examples:

DO-implied List	Equivalent Lists
(X(I), I = 1,4)	X(1),X(2),X(3),X(4)
(A(I), I = 1,10,2)	A(1),A(3),A(5),A(7),A(9)
((C(I,J),D(I,J), J = 1,3), I = 1,4)	C(1,1),D(1,1),C(1,2),D(1,2),C(1,3),D(1,3)
	C(2,1),D(2,1),C(2,2),D(2,2),C(2,3),D(2,3)
	C(3,1),D(3,1),C(3,2),D(3,2),C(3,3),D(3,3)
	C(4,1),D(4,1),C(4,2),D(4,2),C(4,3),D(4,3)
	Since J is the innermost index, it varies more rapidly than I.

SPECIAL LIST CONSIDERATIONS

1. The ordering of a list is from left to right with repetition of items enclosed in parentheses (other than subscripts) when accompanied by controlling DO-implied indexing parameters.
2. An unsubscripted array name in a list implies the entire array.
3. Constants may appear in input/output lists only as subscripts or as indexing parameters.
4. For input lists the DO-implying index parameters (v, e_1, e_2, e_3) may not appear within the parantheses as list items For example, as an input list

(I,J,A(I), I = 1,J,2)	is not allowed
I,J,(A(I), I = 1,J,2)	is allowed

   As an output list

(I,J,A(I), I = 1,J,2)	is allowed

5. The number of items in a single list is limited only by the statement length specifications.

INPUT/OUTPUT STATEMENTS

All input/output statements specify a device unit number, u. This number may be either an integer constant or an integer variable reference whose value then identifies the unit. This unit number corresponds to an actual physical device in one of two ways:

1. The number may be assigned to a device at program run time.
2. The number may be a standard unit number assignment, which is recognized as referring to a particular device. These standard assignments may be overridden by run-time assignments, if necessary.

Table 6 shows standard device assignments for Sigma FORTRAN IV-H. There are no standard unit assignments for magnetic tapes or random access devices.

Table 6. Standard Unit Assignments

Unit Number	Standard Assignment
101	Typewriter input
102	Typewriter output
103	Paper tape reader
104	Paper tape punch
105	Card reader
106	Card punch
108	Line printer

FORMATTED INPUT/OUTPUT STATEMENTS

Formatted I/O statements are used to process binary-coded (BCD) records. These statements have the forms

```
READ(u, f)k
WRITE(u, f)k
```

where

u	is a device unit number (unsigned integer or integer variable)
f	is a FORMAT statement label or an unsubscripted array name
k	is an input/output list, which may be omitted

A formatted READ statement causes the character string in the external record to be converted, according to the FORMAT specified, into binary values. These are then assigned to the variables appearing in the list k, or the equivalent simple list, if k contains a DO-implication. Conversely, a formatted WRITE statement converts internal values into character strings and outputs them.

Each formatted input/output statement begins processing with a new record. It is not possible to process a particular record using more than one READ or WRITE statement. More than one record may be processed by these statements if specifically requested by the FORMAT statement. However, attempting to read (or write) more characters on a record than are (or can be) physically present does not cause processing of a new record; on output the extra characters are lost, on input they are treated as blanks.

A BCD record has a maximum size of 132 characters. Certain devices may impose other restrictions on the size of records. For example, a punched card contains 80 characters. A record may contain as few as zero characters, in which case it is considered to be blank or empty. In other words, a record into which any number of blanks have been specifically written is indistinguishable (within the program) from an empty record. However, on devices such as magnetic or paper tape, the FORMAT statement may determine the actual size of record written (see the SDS Sigma Monitor reference and operations manuals for a complete description of BCD records).

The list k may be omitted from a formatted input/output statement. Normally, this has the effect of skipping one record (on input) or writing one blank record (on output). However, information may actually be processed, and/or more than one record used, if the FORMAT statement begins with Hollerith or slash specifications, in which case information is either read into or written from the locations in storage occupied by the FORMAT statement (see "H Format Codes" under "FORMAT Statements").

Examples:

```
READ(105,6)X,Y,T(3,5)
READ(5,FORM) (A(I), I = 1,40),H,Q
WRITE(N,FMT)(MASS(J,3),J=1,100,1)
WRITE(102,93) MESAGE, ERR NO
```

ACCEPTABLE FORTRAN II STATEMENTS

The following FORTRAN II statements are accepted by FORTRAN IV-H. Each of these statements designates a specific physical device, as shown in Table 7.

Table 7. FORTRAN II/FORTRAN IV Equivalent Statements

FORTRAN II Statement	FORTRAN IV Equivalent	Standard Assignment
READ f, k	READ (105, f)k	Card reader
PUNCH f, k	WRITE (106, f)k	Card punch
PRINT f, k	WRITE (108, f)k	Line printer

READ Statement

This FORTRAN II input statement has the form

    READ f, k

where

f	is a statement label or an array name of the FORMAT statement describing the data
k	is an input list as described earlier in this chapter

The READ statement causes the character string in the external record to be read from device 105 and converted, according to the FORMAT specified, into binary values which are then assigned to the variables appearing in the list k, or the equivalent simple list if k contains a DO-implication.

PUNCH Statement

This FORTRAN II output statement has the form

    PUNCH f, k

where

f	is a statement label or an array name of the FORMAT statement describing the data
k	is an output list described earlier in this chapter

This statement causes internal data to be converted into character strings, as specified by the applicable FORMAT statement, and output on device 106.

PRINT Statement

The form of the PRINT statement is

    PRINT f, k

where

f	is a statement label or an array name of the FORMAT statement describing the data
k	is an output list as described earlier in this chapter

The PRINT statement causes internal data to be converted into character strings, as specified by the applicable FORMAT statement, and output on device 108 (see also, "Carriage Control for Printed Output" in this Chapter).

INTERMEDIATE INPUT/OUTPUT STATEMENTS

These statements process information in internal (binary) form and are designed to provide temporary storage on magnetic tapes, discs, and drums. They have the form

    READ(u) k

and

    WRITE(u) k

where

u	is a device unit number
k	is an input/output list, which may be omitted (see below)

The binary READ/WRITE statements process data as a string of binary digits, arranged into words, depending on the size of the items in the list k (see "Allocation of Variable Types", Chapter 7). All the items appearing in the list of a binary READ/WRITE statement are contained in one logical record.

A logical record may consist of several physical records; however, it is treated as a single record, as far as the programmer is concerned. (See The SDS Sigma Monitor reference and operations manuals for a description of the format of intermediate binary information.) This means that the information output by a single binary WRITE statement must be input by one and only one READ statement. It is permissible to read less information than is present in the record. If the input list requests more data from a binary record than is present, an error will occur. There is no limit to the number of items that can be processed by a single READ/WRITE statement, since only one logical record will be read or written, regardless of the amount of data to be transferred.

The records produced by binary WRITE statements do not consist of just the data to be generated. Control words are included in the records to facilitate reading or backspacing the proper number of physical records. Thus, the information produced by an intermediate binary WRITE statement is meant to be read subsequently by a binary READ statement. Other FORTRAN systems will not necessarily interpret the records in the same way. Similarly, binary tapes produced on other machines or by other programs cannot, in general, be input using a binary READ statement.

If the list k is omitted from a binary READ/WRITE statement, a record is skipped, or an empty record is written. Unlike formatted input/output statements, no data transfer can occur in such an operation. If an empty record is written, it can only be processed by a READ statement with no list and, therefore, has little purpose.

Examples:

```
READ(3)E1(K),(M(K,L),L = 1,22)

READ(N) ARRAY

WRITE(MIN)R(J),G(J)

WRITE(3)VALUE
```

END and ERR Forms of the READ Statement

Both the formatted and intermediate binary READ statements may optionally include a specification of action to be performed if an error occurs or an end-of-file mark is read. The statements are written

READ(u, f, END=s_1, ERR=s_2)k

READ(u, END=s_1, ERR=s_2)k

where s_1 and s_2 are each a statement label. Both the END = s_1 and the ERR = s_2 are optional; if both are present, either may appear first.

If an end-of-file mark is encountered during the processing of the READ statement, control will be transferred immediately to statement s_1. If an error occurs, control will be transferred to statement s_2.

NAME LIST Input/Output

With NAMELIST input/output the programmer can input and output numeric data without FORMAT statements. Each input record specifies exactly which variable is being input, rather than requiring the input data to interface with an internal input list. This also provides the ability to input information without knowing in advance which items are going to be processed.

NAMELIST input/output statements have the form

READ(u,x)

WRITE(u,x)

where

u	is a device unit number (unsigned integer or integer variable)
x	is a NAMELIST block name

The NAMELIST block name x is a single variable name that refers to a specific list of variables or array names into which (or from which) the data is transferred. A specific list (block) of names receives a NAMELIST name through a NAMELIST statement. The form of the NAMELIST statement is

NAMELIST w_1 w_2 w_3 ... w_n

where the w_i, which should not be separated by commas, have the form

$/x/ v_1, v_2, v_3, \ldots, v_m$

where

x   is the name of the NAMELIST block

v_j is a scalar or array name and are not allowed to be dummy names

Each of the w_i defines a NAMELIST block into which the names of the variables v_j are placed. The following rules apply to the NAMELIST statement:

1. The name (x) of a NAMELIST block consists of up to six alphanumeric characters, the first of which is alphabetic. It must be unique with respect to all other identifiers in the program, except for statement function dummies.
2. A NAMELIST block can be defined only once. Unlike labeled COMMON (which the NAMELIST statement resembles syntactically) a NAMELIST block cannot be continued in a later NAMELIST statement. Once the block name has appeared in a NAMELIST statement, it may appear only in NAMELIST-type READ and WRITE statements in the rest of the program.
3. A NAMELIST block may appear anywhere in a FORTRAN program but must be defined prior to its appearance in a READ or WRITE statement.
4. A variable may belong to more than one NAMELIST block.

Output Format

The form of WRITE statement used with NAMELIST blocks is

WRITE(u, x)

where

u   is the logical number, as defined for other WRITE statements

x   is a NAMELIST block name

There is no input/output list associated with this form of WRITE statement. The rules defining this form of output are:

1. All the variables belonging to NAMELIST block x are written in the unit specified by u. This relieves the programmer of the necessity of writing a complete input/output list with each WRITE statement.
2. Since only scalar and array names may appear in a NAMELIST block, it is not possible to output array elements individually (only the entire array may be output). Similarly, implied DO loops cannot be used.
3. The values of each list item will be output, each according to its own type, and will be preceded by the character string that defines the item. The character string and the value are separated by an equal sign. Output begins in column 2 in order to avoid automatic carriage control (see "Carriage Control for Printed Output" in this chapter).
4. The output buffer for a NAMELIST block can cccommodate 80 characters. The format specifications (see FORMAT Statements" later in this chapter) provided by the compiler for NAMELIST-type WRITE statements are as follows:

Item Type	Format Specification
logical*4	L1
integer*4	I11
real*4	G12.6
double precision*8	G21.15
complex*8	1H(, G12.6, 1H, G12.6, 1H)
double complex*16	1H(, G21.15, 1H, G21.15, 1H)

Leading blanks are removed in integers. Commas separate entries on a card.

5. The first record output by each NAMELIST type WRITE statement consists of an ampersand (&) in column 2, followed by the NAMELIST block name. Subsequent records contain the values of the variables in the block. The last record produced will contain an ampersand (&) in column 2, followed by the letters END.

6. Since each NAMELIST-type WRITE statement creates an END record, the information produced by one such statement may be read by one and only one NAMELIST-type READ statement. It is not possible to create a single "file" using more than one WRITE statement or multiple "files" using a single WRITE statement, nor is it possible to suppress the block name or the END record from appearing on a listing.

Input Format

The NAMELIST form of READ statement, which is the counterpart of the WRITE statement just described, is written as

READ (u, x, END=k_1, ERR=k_2)

where

u	is a device unit number
x	is a NAMELIST block name
k_1 and k_2	are statement labels

Both END=k_1 and ERR=k_2 are optional. They specify action to be taken upon an end-of-file or error condition, respectively (see "END= and ERR= Forms of the READ Statement").

Input with this form of READ statement is governed by the following rules:

1. The first character of every record is ignored.

2. The first record to be read must be limited to an ampersand (&) in column 2, followed immediately by the same NAMELIST block name that was specified in the READ statement. This redundancy check is provided to assure that the proper data is being read.

3. Only variables belonging to the NAMELIST block (x) specified in the READ statement are permissible in the input data. The appearance of any other variable, whether in another NAMELIST block or not, is an error.

4. Scalars, array elements, and entire arrays may be processed by this statement. The form the data items may take is:

   a. variable name = constant

   b. array name = set of constants (separated by commas). The number of constants must be equal to the number of elements in the array.

   The constants used may be integer, real, double precision, complex, logical, or literal. Each constant must be the same type as the variable it initializes, except for literal constants, which may be used with any type of variable (integer is recommended).

5. When inputting into an entire array, a constant may be repeated k times by preceding it with k*, where k is an unsigned integer. k is called the repeat count, and its usage is similar to that in the DATA statement (see Chapter 7). Note that 3*5 does not mean 15, but rather three constants of value "5".

6. There may be any number of replacements on a record. These must be separated by commas and, except for the last data record preceding the END record, the last replacement on each record must be followed by a dangling comma.

7. Blanks may not be embedded in a constant or a repeat count or variable name, but may be used freely elsewhere in the data record.

8. The last item on each record that contains data must be a constant followed by a comma, except as noted in statement 6. The dangling comma after the final data item is optional.

9. The last record processed by the READ statement must contain an ampersand (&) in column 2, followed immediately by the characters END in columns 3, 4, and 5. No other data may appear in the last record.

Example:

Assume A and LGL are array names

```
REAL        A(52)
LOGICAL     LGL(12,4)
COMPLEX     CPX
NAMELIST    /LIST1/DBL,T,A,J/LIST2/LGL,CPX,A,K,ARY
READ(105,LIST2)
:
:
```

If the input is from punched cards

	Column 2
First card	&LIST1
Second card	T=55E-2,A(3)=4,DBL=2,
Third card	J=-3746,
Fourth card	&END
Fifth card	&LIST2
Sixth card	LGL(12,2)=.FALSE.,LGL(12,4)=.TRUE.,
Seventh card	CPX=(7.32D-2,3),ARY=0,K=0,
Eight card	A=9*5,12
Ninth card	&END

The READ statement causes the first card to be read. Since this does not contain the specified NAMELIST name, cards are read (and ignored) until the next NAMELIST name is encountered. When the sixth card is read, the values FALSE and TRUE are stored in LGL(12,2) and LGL(12,4), respectively. The seventh card is read, and the values for CPX,ARY, and K are stored. Next, the eighth card is read. Since A is an array name without subscripts, the entire array will be filled with the succeeding constants: A(1) through A(9) would contain the value 5; A(10) would contain the value 12.

FORMAT Statements

FORMAT statements are used in conjunction with READ and WRITE statements to specify data conversion methods and/or editing of data as it is transmitted between computer storage and external devices. These statements are nonexecutable and may be placed anywhere in the source program; they must have statement labels for reference by input/output statements.

FORMAT statements have the form

FORMAT $(S_1,S_2,S_3,\ldots,S_n)$ where $n \geq 0$

Each S_i is either a format specification of one of the forms described in the paragraphs below or a repeated group of such format specifications in the form

$r(S_1,S_2,S_3,\ldots,S_m)$

where

r is a repeat count as described on the following page

S_j is a format specification

The word FORMAT and the parentheses must appear as shown. Commas separate the format specifications and must be present as shown or may be replaced by slashes or groups of slashes (see "Format Specifications").

A field is defined as that part of an external record occupied by one transmission item.

FORMAT SPECIFICATIONS

Format specifications describe the size of data fields and specify the type of conversion and editing to be exercised upon each transmitted item. FORTRAN IV-H recognizes twelve format codes:

F	Real floating-point without exponent
D	Double-precision floating-point with exponent
E	Real floating-point with exponent
G	Generalized integer or floating-point
I	Decimal integer
L	Logical
A	Alphanumeric specification
H	Hollerith string
'	Literal specification
X	Blank or skip specification
T	Tab specification
P	Scale factor

Format specifications may be in any of the following forms:

```
rFw.d    rIw    's'    /
rEw.d    rLw    iX
rDw.d    rAw    Tw
rGw.d    nH     iP
```

where

The characters F, E, D, G, I, L, A, H, quotation mark ('), X, T, P, and slash(/) define the type of conversion, data generation, scaling, editing, and FORMAT control.

w is an unsigned integer that defines the total field width in characters (including digits, decimal points, algebraic signs, exponent field, and blanks) of the external representation of the data being processed.

d for F, E, and D specifications, is an unsigned integer that specifies the number of fractional digits appearing in the magnitude portion of the external field.

For G specifications, d is also an unsigned integer, but in this context it is used to define the number of significant digits that appear in the field.

n is an unsigned, decimal integer that defines the number of characters being processed.

s is a string of characters acceptable to the FORTRAN IV-H processor.

i is an integer value; its function is described under X and P specifications

r (repeat count) is an optional, unsigned integer that indicates the specification is to be repeated r times. When r is omitted, its value is assumed to be 1. For example,

3I6

is equivalent to I6,I6,I6

F Format (Fixed Decimal Point)

Form:

rFw.d

Output. The F format code may be used to process real data that does not contain a decimal exponent; w characters are processed, of which d are considered fractional.

Internal values are output as real constants, rounded to d decimal places with an overall length of w. The total field length reserved must include sufficient positions for a sign (if any) and a decimal point. If minus, the sign is printed.

The converted characters are right justified in the field, w, with preceding blanks to fill the field if necessary. If the conversion produces more than w characters, only the rightmost w characters are output. This is not treated as an error. The relationship $w \geq d + 2 + n$, where n is the number of integer digits, must hold true to prevent loss of digits.

Examples:

Format Specification	Internal Value	Output (b indicates blanks)
F5. 2	12. 17	12. 17
F10. 4	368. 42	bb368. 4200
F7. 1	-4786. 361	-4786. 4
F4. 4	-1. 22315	2232
F4. 4	432034.	0000

Input. See description under "Input" for D- and E-type conversion.

D and E Format (Normalized, with Exponent)

Form:

rDw.d

rEw.d

Output. The D and E format codes may be used to process real data. A D format code indicates a word length of 8 bytes. An E format code indicates a word length of 4 bytes.

Internal values are output as real constants, rounded to d digits, in the order given below.

1. a minus sign or blank (if positive)
2. a zero
3. a decimal point
4. d digits
5. the letter D or E
6. the sign of the exponent (minus or blank)
7. a 2-digit exponent (i. e., the power of 10 by which the number must be multiplied to obtain its true value)

The values, as described above, are right justified in the field, w, with preceding blanks to fill the field if necessary. If the conversion produces more than w characters, only the rightmost w characters are output. This is not treated as an error. To ensure against this loss of characters on the left, the relationship $w \geq d + 7$ must be satisfied by the format specification.

Examples:

Format Specification	Internal Value	Output (b indicates blanks)
E12. 5	76. 573	b0. 76573Eb02
D13. 6	-32672. 354	-0. 326724Db05
D13. 7	-32672. 354	0. 3267235Db05
E10. 3	-0. 008	-0. 800E-02
D10. 3	-38. 0068	-0. 380Db02
E11. 4	. 361887	0. 3619Eb00
E8. 4	. 361887	3619Eb00

Input. Input is the same for D, E, and F format codes. Each input string will be of length w with d characters in the fractional portion. If a decimal point is present in the input string, the value of d is ignored, and the number of digits in the fractional portion of the value will be explicitly defined by that decimal point. Leading, embedded, and trailing blanks are treated as zeros.

When input data is read, the start of the exponent field must be marked by an E, or a + or - sign (not a blank). Thus, E5, E+5, +5, +05, E05, and E+05 all have the same effect and are permissible decimal exponents for input.

Examples:

Format Specification	Input (b indicates blanks)	Internal value
E7.1	-36.273	-36.273
F8.3	b1.62891	+1.62891
F8.3	b1628911	+1628.911
E10.3	+0.13756+4	+1375.60
D10.3	bbbbb17631	17.631
E10.3	-763267E-3	-0.763267

During the conversion the decimal point is first positioned according to the format specification. The value of the exponent is then applied to determine the actual position of the decimal point. In the last example, -763267E-3 with a specification of E10.3 is interpreted as -763.267E-3; which, when evaluated (i.e., -763.267×10^{-3}), becomes -0.763267.

G Format (General)

Form:

rGw.d

G is the only format that may be used with any type of data, including logical. The form of conversion it performs depends on the type of the list items. For a Gw.d specification, the following table shows the equivalent format that is used when processing list items of the various types.

List Item Type	Input	Output
integer	Iw	Iw
real	Fw.d	(see below)
double precision	Fw.d	(see below)
logical	Lw	Lw

Note that complex values are processed as two separate items. The real and imaginary parts require individual specifications, and conversion occurs, as shown above, for real or double-precision data.

For integer and logical list items, the d (in Gw.d) need not be specified; if it is present, it will be ignored. This is the only case in which d is not assumed to be zero if not specified.

If the absolute value of the real data (n) is in the range $0.1 \le n < 10^{**}d$ (where d is as in Gw.d), this exponent field is blank. Otherwise, the real data is transferred with an E or D decimal exponent depending on the length specification (either four or eight storage locations, respectively) of the real data. In the first case (i.e., n in the range $0.1 \le n < 10^{**}d$) four blanks are output, following the number, in the positions where an exponent would otherwise be. In this way, numbers that are output in columns will tend to align with each other in a more readable way.

When the width specification w is insufficient to allow expression of the entire value, only w digits will appear. The digits lost are from the left or most significant portion of the field. This is not treated as an error condition. To ensure that such loss will not happen, the following relation should hold true:

$w \ge d + 7$

The following considerations should be kept in mind when this format code is used in a FORMAT statement on input or output:

1. When real data is to be input with a G format code, a decimal point must be included.

2. If the data exists with a D decimal exponent, it is transferred with the D decimal exponent (i.e., it is a double-precision value).

Examples:

1. The following examples illustrate the effect of G format output on values of various sizes:

Value	G10.3	G10.1
.02639	0.264E-01	0.3E-01
.2639	0.264	0.3
2.639	2.64	3.
26.39	26.4	0.3E 02
263.9	264.	0.3E 03
2639.	0.264E 04	0.3E 04

2. Assume that the statements

```
25    FORMAT    (G3,2G9.2,G13.7,2G8.2,G2)
      .
      .
      WRITE    (106,25)INT,R1,R2,R3,CMP,LOG
```

are to be used to output the following data:

Variable	Type	Value
INT	integer, length 4	271
R1	real, length 4	463.81
R2	real, length 4	71.83
R3	real, length 8	6.8576311
CMP	complex, length 8	(2.1,3.7)
LOG	logical, length 4	.FALSE.

If the output is to the line printer, it would appear as

```
G3    G9.2    G9.2    G13.7    G8.2    G8.2    G2

271b0.46Eb03bb72.bbbbb6.857631bbbbb2.1bbbbb3.7bbbbbF
↑                                                  ↑
Print Position 1                                   Print Position 52
```

(b indicates blanks)

I Format (Integer)

Form:

rIw

The I format code is used in processing integer data.

Output. Internal values are output as integer constants. Negative values are preceded by a minus sign. If the converted value does not fill the specified field, the digits are right justified and preceded by blanks. If the converted integer constant requires more positions than are permitted by the value of the width w, only w digits appear in the external string, and the excess leftmost (i.e., most significant) digits are lost.

Examples:

Format Specification	Internal Value	Output (b indicates blanks)
I6	+281	bbb281
I6	-384	bb-384
I6	-17631	-17631
I4	-17631	7631

Input. A field of w characters are input and converted to internal integer format. A minus sign may precede the integer digits. If a sign is not present, the value is assumed to be positive. If the field width specification w is greater than the number of digits being read into the field, the integer data is right justified and preceded by zeros. Leading, embedded, and trailing blanks are treated as zeros.

Examples:

Format Specification	Input(b indicates blanks)	Internal Value
I4	b124	+124
I4	-124	-124
I7	bbb7631	0007631
I5	-31024	-3102

L Format (Logical)

Form:

rLw

Only logical data may be processed with this form of conversion.

Output. Logical values are converted to either a T or an F character for the values "true" and "false", respectively. The T and F characters are preceded by w - 1 blanks.

Examples:

Using the specification L4

.TRUE.	is converted to	bbbT
.FALSE.	is converted to	bbbF

(b indicates blanks)

Input. The first T or F encountered in the next w characters determines whether the value is "true" or "false", respectively. If no T or F is found before the end of the field, the value is "false". Thus, a blank field has the value "false". Characters appearing between the T or F and the end of the field are ignored.

Examples:

The following input fields, processed by an L7 format, have the indicated values:

Input Field	Value	Input Field	Value
T	True	RIGHT	True
TRUE	True	READ	False
.TRUE.	True	STAFF	True
F	False	LEFT	False
FALSE	False	24T+T42	True
.FALSE.	False	bbbb	False

(b indicates blank.)

A Format (Alphanumeric)

Form:

rAw

The A format code is used to read or write alphanumeric data. If w is equal to the number of characters corresponding to the length specification of the items in the I/O list, w characters are read or written.

Output. Internal binary values are converted to character strings at the rate of eight binary digits (two hexadecimal digits) per character. The most significant digits are converted first; that is, conversion is from left to right. The number of characters produced by an item depends on the number of words of storage allocated for that type of item (see "Allocation of Various Types" in Chapter 7). As with all other format conversions, complex data are treated as two real or as two double-precision values. Normally, alphanumeric information is used with integer variables.

When the magnitude of w does not provide for enough positions to express the data completely, the external field is shortened from the right (least significant) end. This is not treated as an error condition. When w has a value greater than necessary, the external character string is right justified in the field and preceded by the appropriate number of blank characters.

Examples:

Data Type	Internal Binary/Hexadecimal	Aw	External String
integer (4), real (4), or logical (4)	1100 1001 1101 0101 1110 0011 0101 1100 C 9 D 5 E 3 5 C	A4	INT*
		A2	IN
		A6	bbINT*
double precision	1100 0100 1101 0110 1110 0100 1100 0010 C 4 D 6 E 4 C 2	A8	DOUBLE=2
		A6	DOUBLE
	1101 0011 1100 0101 0111 1011 1111 0010 D 3 C 5 7 B F 2	A11	bbbDOUBLE=2

(b indicates blanks)

In both of the above examples, the first A format specifies exactly the number of characters required to express the data completely.

Input. When the width w is larger than necessary (that is, when its magnitude is greater than the number of characters associated with the data type of the corresponding list item), the number of characters equal to the difference between w and the length specification are skipped and the remaining characters are read. For example, if the list item is integer (length of 4) and the format specification A10 is used

ABCDEFGHIJ   is converted to   GHIJ

The first six characters are skipped because the difference between w (10) and the length specification (4) is 6.

When the value of w is less than the number of characters associated with the data type of the list item, the most significant positions of the list item are filled with w characters, and the remainder of the positions are filled with blanks. For example, if the list item is double precision and the format specification is A6

UVWXYZ   is converted to   UVWXYZbb

(b indicates blanks)

H Format (Hollerith)

Form:

nHs

where n $\leq$ 255. This descriptor causes Hollerith information to be read into, or written from, the n characters (s) following the nH descriptor in the format specification itself.

Blanks are significant with the H format code and must be included as part of the count n.

Output. The n characters (s) are output to the specified external device. Care should be taken that the character string s contains exactly n characters so that the desired external field will be created and subsequent data will be processed correctly.

Examples (b indicates blanks):

Specification	Output
1HR	R
8HbSTRINGb	bSTRINGb
12HXb(1,3)=12.0	Xb(1,3)=12.0

Input. The n characters of the string s are replaced by the next n characters of the input record. This results in a new string of characters in the format specification.

Examples (b indicates blanks):

Specification	Input Characters	Resultant Specification
5HX=123	X=777	5HX=777
5HTRUEb	FALSE	5HFALSE
9Hbbbbbbbbb	bMATRIXbZ	9HbMATRIXbZ

' Format (Literal)

There is an alternate format for Hollerith transmission that has the advantage of not requiring the characters in the string to be counted.

Form:

```
's'
```

Literal data consists of a string (s) of not more than 255 characters enclosed in single quotation marks. Any characters from the set of acceptable characters (see Chapter 1) may appear in the string; however, a single quotation mark (or apostrophe) must be represented by two consecutive ' characters. For example, the statement

```
22     FORMAT     ('ABC''DEF')
```

produces the external string

```
ABC'DEF
```

Output. The string s is transmitted to the external device in a manner similar to that for H format. Thus,

```
'ABLE' , 'BODIED'
```

is output as the string

```
ABLE BODIED
```

Input. The characters appearing between the single quotation marks are replaced by the same number of characters, taken sequentially from the input string.

Examples (b indicates blanks):

Format Specification	Input Characters	Resultant Specification
'VECTOR'	MATRIX	'MATRIX'
'MAYbb'	JUNEb	'JUNEb'

X Specification (Space)

Form:

```
iX
```

where i is an unsigned integer constant not greater than 255. This specification causes no conversion to occur, nor does it correspond to items in an input/output list. When used for output iX causes i blanks to be inserted in the output record. For input, iX causes i characters of the input record to be skipped.

Output Example:

The specifications

```
'WXYZ', 4X, 'IJKL'
```

generate the external string (b indicates blanks)

```
WXYZbbbbIJKL
```

Input Example:

With the specifications

```
F5.3, 6X, I3
```

and the input string

```
76.41IGNORE697
```

the characters

```
IGNORE
```

will not be processed.

T Specification (Tab)

Form:

```
Tw
```

This specification causes processing, either input or output, to begin at character position w ($w \leq 132$) in the record, regardless of the position in the record that was being processed before the T specification.

Output Example:

```
14  FORMAT  (T21, 'STRESS ANALYSIS',T1, ' REPORT 1',T45, 'SEPT. ''66')
```

This FORMAT statement would produce the following printed line:

```
REPORT  1           STRESS ANALYSIS   SEPT. '66
Print Position 1    Print Position 21   Print Position 45
```

Backward tabbing can cause previously output information to be overprinted or previously read input to be processed again. However, it is not possible to tab to a position that precedes the beginning of the record.

Input Example:

The statement

```
9      FORMAT      (T18, A10)
```

when used for input, would cause the first 17 characters of input data to be skipped and the next 10 characters to be read according to the format specification A10.

P Specification (Scale Factor or Power of 10)

Form:

```
iP
```

(i is a signed integer constant). A P specification causes the value of the scale factor to be set to i, where the scale factor is treated as a multiplier of the form

$$\text{external quantity} = \text{internal quantity} \times 10^{i}$$

Before any processing occurs, the scale factor is set to zero at the beginning of each formatted input/output operation. Any number of P specifications may be present in a FORMAT statement, thereby causing the value of the scale factor to be changed several times during a formatted input/output operation. After a scale factor has been specified, it is effective for all applicable format codes (see below) following the scale factor within the same FORMAT statement. This also applies to format codes enclosed within an additional pair of parentheses. Once a scale factor has been established for an I/O operation, it can be reset to zero in the same FORMAT statement only by use of a 0P specification.

Scale factors are effective only with F, E, and D conversions, input G conversions, and E-type output G conversions.

Output. For output, scale factors have effect only on real data. When used with real data having an E or D exponent, a positive scale factor increases the number and decreases the exponent.

Examples:

Format Specification	External field when internal value is 2.71828	External field when internal value is -2.71828
-2PF10.3	.027	-.027
-1PF10.3	.272	-.272
F10.3	2.718	-2.718
1PF10.3	27.183	-27.183
2PF10.3	271.828	-271.828
-2PE14.3	0.003E 03	-0.003E 03
-1PE14.3	0.027E 02	-0.027E 02
E14.3	0.272E 01	-0.272E 01
1PE14.3	2.718E 00	-2.718E 00
2PE14.3	27.183E-01	-27.183E-01

These examples for E conversion are similar to those that would result from D conversion.

Input. For input, scale factors have effect only on real data that does not contain an E or D exponent. For example, if input data is in the form dd.dddd and it is desired to use it internally in the form .dddddd, then the format code used to effect this change is 2PF7.4.

Examples:

The following examples indicate the effect of scaling during an input operation:

External Field	Scale Factor	Effective Value
-71.436	0P	-71.436
	3P	-.071436
	-1P	-714.36
175.8041	0P	175.8041
	1P	17.58041
	-1P	1758.041

/ Specification (Record Separator)

The form of the / specification is

```
/
```

Each slash (/) specified causes another record to be processed. In the case of contiguous slash specifications (i.e., ////.../), since no conversion occurs between each of the slash specifications, records are ignored during input, and blank records are generated during output operations. The same condition can occur when a slash specification and either of the parenthesis characters surrounding the field specifications are contiguous; a slash preceding the final right parenthesis in a FORMAT statement is not ignored.

Output. Whenever a slash specification is encountered, the current record being processed is output, and another record is begun. If no conversion has been performed when the slash is encountered, a blank record is created. The statements

```
   WRITE (5, 10) X, Y
10 FORMAT (F5.3//I13)
```

are processed in the following manner:

1. A record is begun, and X is converted with the specification F5.3.

2. The first slash is encountered, the record containing the external representation of X is terminated, and another record is begun.

3. The second slash is encountered, the second record is terminated, and a third record is started. Since no conversion occurred between the terminations of the first and second records, the second record was blank.
4. The value of the variable Y is converted with the I13 specification, the closing right parenthesis character is encountered, and the third record is terminated.

If a third item Z were added to the output list, as in

```
WRITE (5, 10) X, Y, Z
```

The following additional steps would occur:

5. A fourth record is begun, and Z is converted using the specification F5.3.
6. The first slash is re-encountered, the fourth record is terminated, and a fifth record is begun.
7. Again, the second slash is processed; the fifth record, which is blank, is terminated; and the sixth record is started.
8. Since there are no more list items, the specification I13 is not processed, and a termination occurs. The final or sixth record, which is also blank, is output.

The two statements

```
  WRITE (M,4) X
4 FORMAT (/E12.4/)
```

cause the generation of a blank record, followed by a record containing the value of X (converted by the specification E12.4), followed by another blank record.

Input. The effect of slash specifications during input operations in similar to the effect for output, except that for input, records are ignored in the cases where blank records are created during output operations. For example, the statements

```
  READ (M,4) X
4 FORMAT (/E12.4/)
```

cause a record to be bypassed, a value from the second record to be converted (with the specification E12.4) and assigned to X, and a third record to be bypassed. As with the last example for output, this means that records created with a FORMAT statement containing slash specifications can be input by use of the identical FORMAT statement. This is not true in FORTRAN systems that ignore a final slash.

PARENTHESIZED FORMAT SPECIFICATIONS

Within a FORMAT statement, specifications may be repeated by enclosing them in parentheses, preceded by an optional repeat count in the form

$r(S_1, S_2, S_3, \ldots, S_m)$

where r and S_i are defined previously, and $m \geq 0$. For example, the statement

```
3 FORMAT (3(A4,F3.2,3X),I3)
```

is equivalent to

```
3 FORMAT (A4,F3.2,3X,A4,F3.2,3X,A4,F3.2,3X,I3)
```

In addition to the parentheses required by the FORMAT statement, up to two levels of parentheses are permitted to enable the user to specify repetition of format codes. For example, the statement

```
21 FORMAT (2(G9.2,2(G9.2,F5.3),D16.9),E10.3)
```

would produce printed output of the form

```
G9.2,G9.2,F10.3,G9.2,F5.3,D16.9,G9.2,G9.2,F5.3,G9.2,F5.3,D16.9,E10.3
```

During input/output processing, each repetitive specification and each singular specification is exhausted in turn.

The following are additional examples of repetitive specifications:

```
  34 FORMAT (4X,2(A8,5X,7G3),I4,3(I2,L5))

1125 FORMAT (/F9.7,5(E15.8,/),E15.8)

   8 FORMAT (2(I8,2(3X,F12.9),F12.9),A16)
```

The presence of parenthesized groups within a FORMAT statement affects the manner in which the FORMAT is rescanned if more list items are specified than are processed in the first scan through the FORMAT statement. In particular, when one or more such groups have appeared, the rescan begins with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. A more complete discussion of this process is contained in, "FORMAT and List Interfacing".

FORMAT AND LIST INTERFACING

Formatted input/output operations are controlled by the FORMAT requested by each READ or WRITE statement. Each time a formatted READ or WRITE statement is executed, control is passed to the FORMAT processor. The FORMAT processor operates in the following manner:

1. When control is initially received, a new input record is read, or construction of a new output record is begun.

2. Subsequent records are started only after a slash specification has been processed (and the preceding record has been terminated) or the final right parenthesis of the FORMAT has been sensed. Attempting to read or write more characters on a record than are, or can be, physically present does not cause a new record to begin. On output the extra characters are lost, on input they are treated as blanks.

3. During an input operation, processing of an input record is terminated whenever a slash specification or the final right parenthesis of the FORMAT is sensed, or when the FORMAT processor requests an item from the list and no list items remain to be processed. Construction of an output record terminates, and the record is written on occurrence of the same conditions.

4. Every time a conversion specification (i.e., F, E, D, G, I, L, or A specification) is to be processed, the FORMAT processor requests a list item. If one or more items remain in the list, the processor performs the appropriate conversion and proceeds with the next field specification. If conversion is not possible because of a conflict between a specification and a data type, an error occurs. If the next specification is one that does not require a list item (i.e., H, ', X, T, or /) it is processed whether or not another list item exists. For example, the statement

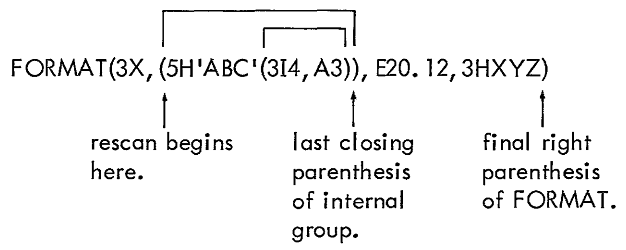
   ```
      WRITE (6,12)
   12 FORMAT(///4HABCD)
   ```

   would produce three blank records and one record containing ABCD before reaching the final right parenthesis. When there are no more items remaining in the list and the final right parenthesis has been reached or a conversion specification has been found, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation.

5. When the final right parenthesis of a FORMAT statement is encountered by the FORMAT processor, a test is made to determine if all list items have been processed. If the list has been exhausted, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation. However, if another list item is present, an additional record is begun, and the FORMAT statement is rescanned. The rescan takes place as follows:

   a. If there are no parenthesized groups of specifications within the FORMAT statement, the entire FORMAT is rescanned.

   b. If one or more parenthesized groups do appear, the rescan is started with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. In the following example, the rescan begins at the point indicated.



```
FORMAT(3X,(5H'ABC'(3I4,A3)),E20.12,3HXYZ)
```

rescan begins here.
last closing parenthesis of internal group.
final right parenthesis of FORMAT.

c. If the group at which the rescan begins has a repeat count (r) in front of it, the previous value of the repeat count is used again for each rescan.

6. Each list item to be converted is processed by one specification or one iteration of a repeated specification, with the exception of complex data. Complex data are processed by two such specifications.

7. Each READ or WRITE statement containing a non-empty list must refer to a FORMAT statement that contains at least one conversion (see step 4 above) specification. If this condition is not met, the FORMAT statement will be processed, but an error will occur.

FORMATS STORED IN ARRAYS

A FORMAT, including the beginning left parenthesis, the final right parenthesis, and the specifications enclosed therein, may be stored in an array. The FORMAT must be stored as a Hollerith string (i.e., a string of characters) usually by use of an input statement.

READ or WRITE statements that refer to a FORMAT stored in an array must reference only the identifier of the array, with no subscripts. For example,

```
WRITE (4,R) E,F,G       refers to a FORMAT stored in an array R.
```

If the variable M is an integer array, the following method may be used to store a FORMAT in M:

the external string

```
(F8.5,4HE9.2,I3)
```

and the statements

```
   READ (N,90)(M(I),I=1.4)
90 FORMAT (4A4)
```

Care must be taken when storing into an array a FORMAT containing specifications of the nHs and 's' forms. In these cases, all characters in the string s, including blank characters, are significant. In all other operations blank characters are insignificant. For example, if M in the above READ statement were double precision instead of integer, the following results would occur:

Element	Storage after READ
M(1)	(F8.bbbb
M(2)	5,4Hbbbb
M(3)	E9.2bbbb
M(4)	,I3)bbbb

which is not the desired result.

Even though a FORMAT may be quite short, such as

```
(I8)
```

it must be stored in an array, and it must not be stored in a scalar variable.

FORMATs stored in arrays may be used by all statements that reference a FORMAT statement.

AUXILIARY INPUT/OUTPUT STATEMENTS

The following set of statements enable the user to manipulate magnetic tapes and sequential disc or drum files.

REWIND Statement

This statement is expressed as

REWIND i

where i is an unsigned integer constant or integer variable.

Execution of a REWIND statement causes the unit whose logical unit number is i to be rewound.

BACKSPACE Statement

The BACKSPACE statement has the form

BACKSPACE i

where i is an unsigned integer constant or integer variable.

When a BACKSPACE statement is executed, the unit referenced by the integer value i is backspaced one logical record. For binary tapes, a logical record may consist of more than one physical record. In this case a logical record is interpreted as all the information output by one binary WRITE statement.

REWIND and BACKSPACE statements that are executed for tapes already positioned at "load point" have no effect.

END FILE Statement

This statement causes end-of-file marks to be written on the specified unit, and has the form

END FILE i

where i is an unsigned integer constant or integer variable whose value determines the unit on which an end-of-file mark is to be written.

Sometimes, it is desirable to take a program that has been written for output on magnetic tape and assign that logical unit number to some other device, such as a line printer. Since such programs often write end-of-file and rewind their tapes at the end of the job, it is permissible to specify an ENDFILE or REWIND operation on any device; the monitor will recognize this anomaly and handle the situation appropriately. It is not permissible to BACKSPACE such devices.

CARRIAGE CONTROL FOR PRINTED OUTPUT

The first character in an output record that is intended for printing may control the printer carriage by containing certain characters:

Character	Effect
1	skip to first line of page before printing
0	space two lines before printing

If one of these characters is present, it is replaced by a blank before the record is printed. The record is not shifted left one position. For example, the 2nd character is printed in column 2.

Any other character appearing as the first character in a record causes the carriage to be single spaced before the record is printed; the record remains unchanged. This includes the "+" character, whose traditional function (overprinting) cannot be performed without hampering the printing speed on all lines.

7. DECLARATION STATEMENTS

Declaration statements are used to define the data type of variables and functions, the dimensions of arrays, storage allocation, initial values of variables, and to provide similar information.

CLASSIFICATION OF IDENTIFIERS

An identifier may be classified as referring to any of the following:

scalar

array

subprogram

COMMON block

The category into which an identifier is placed and the type (if any) associated with it depend on the contexts in which the identifier appears in the program. These appearances constitute explicit or implicit declarations of the way the identifier is to be classified.

IMPLICIT DECLARATIONS

Unless specifically declared to be in a particular category or type, identifiers that appear in executable or DATA statements are implicitly classified according to the following set of rules.

1. When applicable, an identifier is integer if it begins with I, J, K, L, M, or N. It is real if it begins with any other letter (implicit type classification may be altered by use of the IMPLICIT statement).
2. An identifier that is called with a CALL statement is a subprogram.
3. An identifier is a function subprogram if it appears in an expression, followed by an argument list enclosed in parentheses. This does not apply to declared arrays.
4. An identifier is a statement function definition if it appears to the left of an equal sign, followed by a dummy list enclosed in parentheses. It must also comply with the rules given in Chapter 8 under "Statement Functions"; otherwise, it is an error. Again, this does not apply to declared arrays.
5. An identifier is classified as a scalar variable if it makes any other appearances within an executable or DATA statement (i.e., other than followed by a left parentheses or in a CALL statement).
6. An identifier is implicitly classified as a scalar if it does not appear in an executable or DATA statement, but does appear in a COMMON, EQUIVALENCE, or NAMELIST statement.
7. Library functions have an inherent type associated with them, as shown in Table 6 (see Chapter 8). Inherent type is not equivalent to implicit type. Chapter 8 contains a complete description of these functions.

EXPLICIT DECLARATIONS

All other declarations are explicit declarations. Explicit declarations are required in order to classify an identifier in any way other than those described above. Explicit declarations include

array declarations

type declarations

storage allocation declarations

subprogram declarations

subprogram definitions

Explicit declarations override implicit declarations. They may appear anywhere in the program, but must precede the first use (in an executable or DATA statement) of the identifiers appearing in them.

CONFLICTING AND REDUNDANT DECLARATIONS

Except where specifically noted to the contrary, definitions and declarations of the classification of an identifier may not conflict. For example, an identifier may not be both a subprogram name and an array name, both integer and real type, or defined as a subprogram in more than one place, etc.

ARRAY DECLARATIONS

Array declarations explicitly define an identifier as the name of an array variable and have the form

$v(d_1, d_2, d_3, \ldots, d_n)$

where

v is the identifier of the array

n is the number of dimensions associated with the array

d_i is an unsigned integer that defines the maximum value of the corresponding dimension. Arrays may have up to seven dimensions (see "Arrays" in Chapter 3). When v is a dummy array in a subprogram, d_1 through d_n may be scalar variables instead of integers (see "Adjustable Dimensions" in Chapter 8).

Array declarations may appear in

DIMENSION statements

Explicit type statements

COMMON statements

Examples:

X (10)

ARRAY (5, 15, 10)

CUBE (4,7)

DATA (4, 3, 6, 12)

ARRAY STORAGE

Although an array may have several dimensions, it is placed in storage as a linear string. This string contains the array elements in sequence (from low address storage toward high address storage), such that the leftmost dimension varies with the highest frequence, the next leftmost dimension varies with the next highest frequence, and so forth (i.e., 2-dimensional arrays are stored "column-wise"). Figure 2 illustrates array storage.

array A(3, 3, 2)	
Item	Element
1	A(1, 1, 1)
2	A(2, 1, 1)
3	A(3, 1, 1)
4	A(1, 2, 1)
5	A(2, 2, 1)
6	A(3, 2, 1)
7	A(1, 3, 1)
8	A(2, 3, 1)
9	A(3, 3, 1)
10	A(1, 1, 2)
11	A(2, 1, 2)
12	A(3, 1, 2)
13	A(1, 2, 2)
14	A(2, 2, 2)
15	A(3, 2, 2)
16	A(1, 3, 2)
17	A(2, 3, 2)
18	A(3, 3, 2)

Figure 2. Array Storage

REFERENCES TO ARRAY ELEMENTS

References to array elements must contain the number of subscripts corresponding to the number of dimensions declared for the array (except as discussed for EQUIVALENCE statements). References that contain an incorrect number of subscripts are treated as errors.

Furthermore, the value of each subscript should be within the range of the corresponding dimension, as specified in the array declaration. Otherwise, the references may not be to data belonging to the set of elements that comprise the array.

DIMENSION Statement

This statement is used only to define the dimensions of arrays, and has the form

DIMENSION $v_1, v_2, v_3, \ldots, v_n$

where the v_i are array declarations as described previously. A DIMENSION statement does not affect the type or allocation of the arrays declared. For example:

```
DIMENSION MGO(17), LTO(15), BB(36, 22, 34)

DIMENSION AD(184), X(2, 3, 4, 5, 10), PETROL(5, 6)
```

IMPLICIT Statement

This statement is used to alter the conventions for implicit typing from the IJKLMN rule discussed under "Implicit Declarations". It has the form

IMPLICIT $C_1, C_2, C_3, \ldots, C_n$

where

each C_i is a type convention of the form

$type(c_1, c_2, c_3, \ldots, c_m)$

and type is one of the four type declarations:[t]

INTEGER
REAL
COMPLEX
LOGICAL

c_j is a single alphabetic character or two such characters separated by a dash (minus sign); the second character must follow the first in alphabetic sequence. For example,

Z, A-G, M-N, S

This statement, which must appear before the first executable statement and the data statement in the program, causes identifiers beginning with the characters or ranges of characters specified to be implicitly classified with the type specified. An IMPLICIT declaration may override the normal (IJKLMN) rule of implicit type classification. It, in turn, may be overridden by an explicit type declaration (see below). As an example, the statement

```
IMPLICIT COMPLEX(C), LOGICAL(T, F, L-N), INTEGER(H-J, W)
```

would cause the following implicit type conventions to be in force:

1. Identifiers beginning with C are complex.
2. Identifiers beginning with T, F, L, M, or N are logical.
3. Identifiers beginning with H, I, J, or W are integer. The I and J are redundant here, because these are normally integer.
4. Identifiers beginning with K are integer (normal convention).
5. All other identifiers are real (normal convention).

The statement

```
IMPLICIT REAL(A-Z)
```

would cause all identifiers to be real unless explicitly declared otherwise.

[t] "Optional Size Specifications" later in this chapter describes the declaration of double-precision and double-complex types.

While an implicit type declatation may be redundant, it must not conflict with any other implicit type declaration. For example, the statement

```
IMPLICIT REAL(A-Z)  ,  INTEGER(N)
```

is illegal because N is declared to be both real and integer.

An IMPLICIT statement does not affect the types of basic external library functions.

EXPLICIT TYPE STATEMENTS

These statements are used to define, explicitly, the type of an identifier. They have the form

type $S_1, S_2, S_3, \ldots, S_n$

where

type is one of the declarations[t]

INTEGER

REAL

DOUBLE PRECISION

COMPLEX

LOGICAL

S_i is a type specification that is either the identifier of a scalar, array, function, or is an array declaration. Optionally, a scalar, array, or array declaration may be followed by a DATA constant list enclosed in slashes, for the purpose of defining initial values for the variables. In other words, each type specification may take any of the following forms:

identifier

array declaration

identifier/DATA constant list/

array declaration/DATA constant list/

For a description of DATA constant lists, and their function, see "DATA Statement" later in this chapter.

Note that

```
REAL X,Y,Z/3.7/
```

initializes only Z, while

```
DATA X,Y,Z/3.7,3.7,3.7/
```

initializes X, Y, and Z.

Examples of explicit type statements:

```
COMPLEX C3,ALPHA,CARRY(5,5), XYZ

LOGICAL BINARY, BOOLE(4,4,4,4), TRUTHF

INTEGER GEORGE, NETRTE(9)/0,1,1,2,3,5,8,13,21/,MASS/0/

INTEGER ROOT, PP
```

[t]See also "Optional Size Specification" in this chapter.

An explicit type declaration overrides any implicit declaration. Thus, the statements

```
IMPLICIT LOGICAL(L-P)
REAL LEVEL, PERCNT
```

in combination with the standard implicit typing rule, would cause the following identifers to have the types indicated:

LEVEL3	-	logical
LEVEL	-	real
KAPPA	-	integer
POROUS	-	logical
PERCNT	-	real
X	-	real

Type statements may appear anywhere in the program before the first data statement; however, they must precede the first use (in an executable statement) of the identifiers specified.

OPTIONAL SIZE SPECIFICATIONS

In addition to the standard type declarations, an optional form is provided that specifies the exact size of the data. This option takes the form

```
*n
```

where n is the number of bytes occupied by the data (there are four bytes in a word, and eight bits in a byte). In the case of integer and logical, only the standard size is permitted, and the option has no effect. However, this option is used to change real to double precision and complex to double complex, as shown below.

Type	Standard Size (bytes)	Optional Size (bytes)
Integer	4	–
Real	4	8
Complex	8	16
Logical	4	–

Double precision data are identical to real data with size specification of 8 bytes; double complex data are identical to complex data with size specification of 16 bytes. Thus,

INTEGER*4	=	INTEGER
REAL*4	=	REAL
REAL*8	=	double precision
COMPLEX*8	=	COMPLEX
COMPLEX*16	=	double complex
LOGICAL*4	=	LOGICAL

The *n modifier may appear in three kinds of statements: IMPLICIT statements, FUNCTION statements (discussed in Chapter 8), and explicit type statements. This position of the *n relative to the type declaration that it modifies, depends on the statement, as follows:

1. In the IMPLICIT statement, the *n is appended to the type declaration word, as in

```
IMPLICIT     REAL*8(I-K), INTEGER*4(A-H), LOGICAL(L,N)
```

2. In the FUNCTION statement, the *n is appended to the name of the function, rather than to the type word.

```
REAL FUNCTION MULT*8(X, Y, Z)
COMPLEX FUNCTION CNVERT*16(C)
```

3. In explicit type statements, the *n can be appended to the type word, or the identifiers being declared, or both. When appended to the type word, the *n holds for all identifiers listed, excepting those with an individual size specification of their own. In other words, the *n appended to an identifier takes precedence over the *n applying to the whole statement. For example:

```
COMPLEX*8 CUM, LAUDE*16
LOGICAL FLAG(10), TRUTH*4(10)
```

In the first example CUM and LAUDE are both of type complex; CUM has 8 bytes, while LAUDE has 16. In the second example FLAG and TRUTH are arrays, each having 10 elements. Four bytes are required for each element of array FLAG, and 4 bytes per element are required for array TRUTH.

STORAGE ALLOCATION STATEMENTS

These statements are used to arrange variable storage in special ways, as required by the programmer. If no storage allocation information is provided, the compiler allocates all variables within the program in an arbitrary order. The storage allocation statements are

COMMON statement

EQUIVALENCE statement

To make proper use of the storage allocation statements, it is often necessary to know the amount of storage required by each type of variable. The following table indicates the standard size associated with each type.

Type	Words
integer	1
real	1
double precision	2
complex	2
double complex	4
logical	1

COMMON Statement

The COMMON statement is used to assign variables to a region of storage called COMMON storage. COMMON storage provides a means by which more than one program or subprogram may reference the same data.

The COMMON statement has the form

COMMON $w_1\ w_2\ w_3 \cdots w_n$

where

the w_i have the form

$/c/v_1, v_2, v_3, \ldots, v_m$

where

c	is either the identifier of a labeled COMMON block or is absent, indicating blank COMMON
v_i	is a scalar, array name, or array declaration

When w_1 (the first specification in the statement) is to specify blank COMMON, the slashes may be omitted. In all other places, blank COMMON is indicated by two consecutive slashes. For example:

```
COMMON MARKET, SENSE /GROUP3/X, Y, JUMP // GHIA, COLD
```

For each specification (w_i), the variables listed are assigned to the indicated COMMON block or to blank COMMON. The variables are assigned in the order they appear. Thus, in the above example, MARKET, SENSE, GHIA, and COLD are assigned to blank COMMON, while X, Y, and JUMP are placed in labeled COMMON block GROUP3.

Labeled COMMON

Labeled COMMON blocks are discrete sections of the COMMON region and, as such, are independent of each other and blank COMMON.

Any labeled COMMON block may be referenced by any number of programs or subprograms that comprise an executable program (see Chapter 8). References are made by block name, which must be identical in all references. All labeled COMMON blocks need not be defined in any one program; in fact, only those blocks containing data needed by the program require definition.

The variables defined as being in a particular labeled COMMON block do not necessarily have to correspond in type or number between the program in which the block is referenced. However, the definition of the overall size of a labeled COMMON block must be identical in all the programs in which it is defined. For example:

```
SUBROUTINE A                 SUBROUTINE B
REAL T, V, W, X(21)          COMPLEX G, F(11)
COMMON /SET1/T,V,W,X         COMMON/SET 1/G,F
  .                            .
  .                            .
  .                            .
```

Both references to the COMMON block, SET 1, correspond in size. That is, both subprograms define the block SET1 as containing 24 words; the definition in subroutine A specifies 24 items of real type, and the definition in subroutine B declares 12 items of complex type.

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

Block names must be unique with respect to:

1. Subprogram names defined, explicitly or implicitly, to be external references

2. Other block names

A labeled COMMON block may have the same name as an identifier in any classification other than those above; however, it is usually preferable to choose block names that are totally unique.

Blank COMMON

The section of the COMMON region assigned to blank (or unlabeled) COMMON is not discrete; in other words, there is only one such area, and empty block name specifications always refer to it. Furthermore, as opposed to labeled COMMON, blank COMMON areas, defined in the various programs and subprograms that comprise an executable program (see Chapter 8), do not have to correspond in size. For instance, the following two subprograms define blank COMMON areas of different sizes, and yet both may be portions of the same executable program.

```
SUBROUTINE GAMMA             SUBROUTINE ETA
COMMON E,D(20,10),S          COMMON R(10),N(5)
  .                            .
  .                            .
  .                            .
```

Subroutine GAMMA defines a minimum of 202 words in blank COMMON; subroutine ETA declares blank COMMON that contain a maximum of 60 words, depending on the types of the variables E, D, S, R, and N.

Any number of references may be made to blank COMMON with a program. The multiple references may occur in a single COMMON statement or in several COMMON statements. In either case, all variables defined as being in blank COMMON will be placed together in the blank COMMON area.

Variables in blank COMMON may not be initialized (using a DATA statement) while those in labeled COMMON may (see "DATA Statement" later in this section).

Arrangement of COMMON

Each labeled COMMON block and the blank COMMON area contain, in the order of their appearance, the variables declared to be in the labeled block or the unlabeled area. The variables in each section of the COMMON region are arranged from low address storage toward high address storage. The first variable to be declared as being in a particular section is contained in the low address word or words of that section. Array variables are stored in their normal sequence (see "Array Storage") within the COMMON block or area. For example the statements:

```
    COMMON /E/W,X(3,3) //T,B,Q /E/J
    COMMON K,M /E/Y//C(4),H,N(2),Z
```

cause the following arrangement of COMMON:

Item	Block E	Blank COMMON
1	W	T
2	X(1, 1)	B
3	X(2, 1)	Q
4	X(3, 1)	K
5	X(1,2)	M
6	X(2,2)	C(1)
7	X(3,2)	C(2)
8	X(1,3)	C(3)
9	X(2,3)	C(4)
10	X(3,3)	H
11	J	N(1)
12	Y	N(2)
13		Z

Since a segment of the COMMON region may be defined differently in each program, it may be quite important to be aware of which items in a segment contain certain variables. For example,

```
    SUBROUTINE TOM           SUBROUTINE DICK        SUBROUTINE HARRY
    COMMON /S/A,B(101)       COMMON /S/A,X(51)      COMMON /S/ALPHA(52)
                             COMMON /S/Y(50)        COMMON /S/Y(50)
    .                        .                      .
    .                        .                      .
    .                        .                      .
```

will define the block S as follows:

Item	TOM	DICK	HARRY
1	A	A	ALPHA(1)
2	B(1)	X(1)	ALPHA(2)
3	B(2)	X(2)	ALPHA(3)
⋮	⋮	⋮	⋮
52	B(51)	X(51)	ALPHA(52)
53	B(52)	Y(1)	Y(1)
54	B(53)	Y(2)	Y(2)
⋮	⋮	⋮	⋮
102	B(101)	Y(50)	Y(50)

which allows the routine TOM and DICK to access the variable A by that identifier, the routines DICK and HARRY to access the array variable Y by that identifier, and yet the integrity of the block S is maintained (these examples assume A, B, X, Y, and ALPHA are of the same type).

Referencing of Data in COMMON

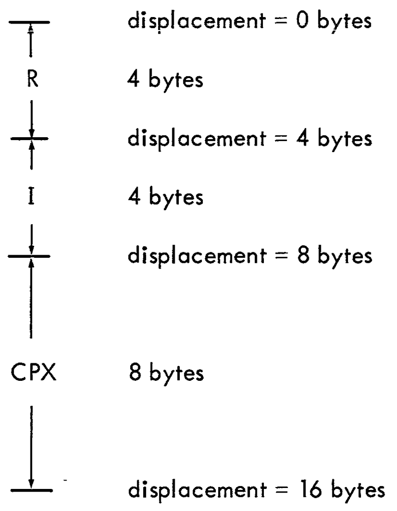
Incorrect referencing of COMMON data will terminate execution. To ensure correct referencing of data, COMMON blocks must be constructed so that the displacement of each variable in the block is an integral multiple of the reference number associated with the variable (displacement is the number of bytes from the beginning of the block to the first storage location of the variable). The reference number for type of variable is shown in the following chart:

Type of Variable	Reference Number
Integer	4
Real	4
Double Precision	8
Complex	8
Double Complex	8
Logical	4

The FORTRAN IV-H system automatically begins every COMMON block as if its specification were 8, thus allowing a variable of any length to be the first assigned within a block. To obtain the correct displacement for other variables in the same block, it may be necessary to insert an unused variable in the block. For example, if the variables R, I, and CPX are REAL, INTEGER, and COMPLEX, respectively, and a COMMON block is defined as

```
COMMON R,I,CPX
```

the displacement of these variables within the block is as shown below:



The displacements for I and CPX are evenly divisible by their reference numbers. However, if R were REAL*8 (instead of length 4), the displacement of CPX would be 12, which is incorrect. In that case, an extra word with a length of 4 bytes would have to be inserted between R and I or between I and CPX to provide the proper displacement for CPX.

EQUIVALENCE Statement

The EQUIVALENCE statement controls the allocation of variables relative to one another. Generally, it is used to assign more than one variable to the same storage location or locations. It is expressed as

EQUIVALENCE $s_1, s_2, s_3, \ldots, s_n$

where each of the s_i is an equivalence set of the form

$(v_1, v_2, v_3, \ldots, v_m)$

Each equivalence set specifies that all the v_i are to be assigned the same storage location. The v_i may be one of the following three forms:

1. A scalar or array name. For arrays, the location referenced is that of the first element.
2. An array element, where the subscripts are unsigned integers. For example, the statements

   ```
   DIMENSION A(3,3)
   REAL B, C, A, X(11)
   EQUIVALENCE (A(1,3), B), (C, X(1))
   ```

   would make B and A(1,3) equivalent, and, similarly, C and X(1) equivalent.

   When multiple subscripts are to be used in an EQUIVALENCE statement, that statement must be preceded by a DIMENSION statement in which the array is declared.
3. An array name followed by an unsigned integer element count enclosed in parentheses. The meaning of this count is as follows: the location of the first element of the array is denoted as position 1; the element immediately following is position 2; and so on. Thus, if X is a 3 x 3 array, X(1) means the same as X(1, 1); X(3) is two elements beyond X and refers to X(3, 1), where the size (in words) of an element is dependent on the type of X (see "Allocation of Variable Types").

   ```
   REAL B, C, A(3,3)
   EQUIVALENCE (A(7), B)
   ```

   would make A(1,3) and B equivalent.

   See also "Interactions of Storage Allocation Statements", below, for further rules concerning equivalences that cannot be implemented.

   Example:

   The effect of the statements

   ```
   DIMENSION W(3), X(3,3), LC(7)
   REAL W,X
   INTEGER LC, J
   REAL * 8 ELSIE
   COMPLEX C
   EQUIVALENCE (W,LC(2), ELSIE), (X(6), J, C(3))
   ```

   is to cause the indicated equivalences:

Word	Variables – Set 1	Variables – Set 2
1	LC(1)	X(1, 1)
2	LC(2) = W(1) = $ELSIE_1$	X(2, 1) = C_1
3	LC(3) = W(2) = $ELSIE_2$	X(3, 1) = C_2
4	LC(4) = W(3)	X(1,2)
5	LC(5)	X(2,2)
6	LC(6)	X(3,2) = J
7	LC(7)	X(1,3)
8		X(2,3)
9		X(3,3)

where the arrangement of set 1 has no bearing on the arrangement of set 2.

The statement

```
EQUIVALENCE (LC(2),W), (W(1),ELSIE), (C(3),J), (J,X(6))
```

has the same results as the EQUIVALENCE statement in the previous example, and the set (J, X(3,2)) is the same as the set (J, X(6)) in this case.

INTERACTIONS OF STORAGE ALLOCATION STATEMENTS

No storage allocation declaration is permitted to cause conflicts in the arrangement of storage. Each COMMON and EQUIVALENCE statement determines the allocation of the variables referenced in them Therefore, no EQUIVALENCE set should contain references to more than one variable than has previously been allocated. When this is not followed, such references are either redundant or contradictory. The redundancy is normally ignored; the contradictory reference is not allowed.

In all cases, the storage allocation sequence specified in a COMMON statement takes precedence over any EQUIVALENCE specifications. Consequently, EQUIVALENCE statements are not allowed to define conflicting allocations of COMMON storage; that is, two variables in the same COMMON block or in different COMMON blocks can not be made equivalent.

It is permissible for an EQUIVALENCE to cause a segment of the COMMON region to be lengthened beyond the upper bound established by the last item defined to be in that segment. However, it is not permissible for an EQUIVALENCE declaration to cause a segment to be lengthened beneath the lower bound established by the first item declared to be in that segment. Both conditions are demonstrated in the examples below.

```
COMMON /BLK1/A(5), B/BLK2/E(4), H, Y(2,2)

DIMENSION Z(10), V(5)

EQUIVALENCE (A, Z), (V(4), E(2))
```

The first EQUIVALENCE set is a permissible extension of the block BLK1, whereas the second set illegally defines an extension of the block BLK2. The declared storage allocation would appear as shown below.

Item	BLK1	BLK2 (illegal extension)
-		V(1)
-		V(2)
1	A(1) = Z(1)	E(1) + V(3)
2	A(2) = Z(2)	E(2) = V(4)
3	A(3) = Z(3)	E(3) = V(5)
4	A(4) = Z(4)	E(4)
5	A(5) = Z(5)	H
6	B = Z(6)	Y(1, 1)
7	Z(7)	Y(2, 1)
8	Z(8)	Y(1, 2)
9	Z(9)	Y(2, 2)
10	Z(10)	

Note: Assume all items are of the same data type.

The fact that COMMON segments may be lengthened by EQUIVALENCE declarations in no way nullifies the requirement that labeled COMMON blocks of the same name, which are defined in separate programs or subprograms comprising portions of an executable program, contain the identical number of words.

EXTERNAL Statement

The EXTERNAL statement has the form

EXTERNAL $p_1, p_2, p_3, \ldots, p_n$

where the p_i are subprogram identifiers.

The EXTERNAL statement declares, as a subprogram, names that might otherwise be classified implicitly as scalars, so that they may be passed as arguments to other subprograms (see "Arguments and Dummies" in Chapter 8). For example, if the subprogram name F appears in the statement

```
CALL    ALPHA(F)
```

but appears in no other context to indicate that it is a subprogram, it would be implicitly classified as a scalar. The EXTERNAL statement can be used to avoid this.

Examples:

```
EXTERNAL    ABS, DABS
CALL    COMPRE (ERROR, ABS, DABS, X)
```

In this example the subprogram identifiers ABS and DABS are used as arguments in the subprogram COMPRE.

```
CALL    SUBR (ERROR, ABS(X, Y), ALPHA, X)
```

In this example the subprogram named ABS is not an argument; it is executed first, and its result becomes the argument. In this situation an EXTERNAL statement is not required.

BLOCK DATA Subprograms

SDS FORTRAN IV-H permits variables in labeled COMMON to be initialized in a special program called a BLOCK DATA subprogram, which begins with a statement of the form

```
BLOCK DATA
```

and may contain only declaration statements (described in this chapter) and DATA statements described below. The subprogram must be terminated with an END statement. Since BLOCK DATA subprograms may not be called by other programs, they have no names nor are they executed in the usual sense.

Example:

```
BLOCK DATA
COMMON    /BLK1/A,B,C,D
REAL    B(5,5)/25*9300./,D/3765.)
COMPLEX A/(4.3,2.4)/
END
```

When initializing variables in labeled COMMON, complete declarations should be included for all the variables in each COMMON block, so that:

1. The position within the block of those variables that are being initialized will be correctly established.
2. The size of each COMMON block will correspond to the size declared in all other programs that use it.

Data may be entered into more than one COMMON block in a single BLOCK DATA subprogram.

DATA Statement

The DATA statement has the form

DATA S_1, S_2, S_3,...,S_n

where

S_i is a data set specification of the form

variable-list/constant-list/

The primary purpose of the DATA statement is to give names to constants: for example, instead of referring to π as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, if a more accurate value is required.

Giving PI a value with a DATA statement is somewhat different from giving it a value with an assignment statement. With the DATA statement the value is assigned when the program is loaded; with the assignment statement, PI receives its value at execution time.

Consider another example that profits even more from the use of the DATA statement: An ARCTAN function can be written using a power series expansion. The efficient way to program this in FORTRAN is with a DO loop, stepping through the constants. But constants cannot be subscripted, and the timing of the routine is adversely affected if an array must be initialized each time into the routine using assignment statements, such as:

```
C(0) = 0
C(1) = .12435 49945
C(2) = .24477 86631
etc.
```

Here, the DATA statement can be used to great advantage. It is not recommended that the DATA statement be used to give "initial" values to variables that are going to be changed. This causes proper initialization of the program to depend on loading and disallows restarting the program once it has changed these values. Good programming practice dictates that such initialization be done with executable statements, e.g., with assignment statements.

The effect of the DATA statement is to initialize the variables in each data set to the values of the constants in the set, in the order listed. For example, the statement

```
DATA X,A,L/3.5,7,.TRUE./, ALPHA/0/
```

is equivalent to the assignment statements

```
X = 3.5
A = 7
L = .TRUE.
ALPHA = 0
```

except that the DATA statement is not executable; its assignments take place upon loading.

Variable and constant lists in DATA statement may be constructed as described in the following two sections.

DATA Variable List

A DATA variable list is similar to an input list (see Chapter 6), in that it may contain scalars or subscripted or unsubscripted arrays. It may not contain implied DO loops. Subscripts must be integers.

DATA Constant List

A DATA constant list is of the form

$c_1, c_2, c_3, \ldots, c_m$

where

the c_j are either constants or repeated groups of constants in the following forms:

c

r*c

where

c	is a signed or unsigned constant of an appropriate type (see below)
r	is an unsigned integer repeat count, whose value (nonzero) indicates the number of times the group is to be repeated

The constant may be any of the forms described in Chapter 2, including literal constants.[†] The type of the constant must be the same as the type of the variable that it is initializing. The following rules apply in DATA statements:

1. Integer, real, double-precision and complex variables may be initialized with constants of those types.

2. Logical constants may be expressed as .TRUE. and .FALSE. or abbreviated as T and F.

3. Literal constants may be used with any type of variable, although integer is recommended. A literal constant is broken up on a character-by-character basis and depends on the number of words of storage occupied by the variable (see "Allocation of Variable Types", Chapter 7). That is, an integer variable requires 4 characters, a complex variable -8 characters, and a double-complex variable, 16 characters.

   Variable items will be initialized as required to use up the characters specified. If there are insufficient characters in any literal constant to fill the last variable used, it will be filled out with trailing blanks.

4. A constant may not be used for more than one variable list item.

The following examples illustrate some of the features described above:

```
INTEGER MM(3)
COMPLEX C1,C2
DATA MM/'ABCDEF', 'GH'/,C1,C2/(17.8,-4.0),(17.8,-4.0)
```

The above DATA statement causes the following assignments to be made:

```
MM(1) = 4HABCD
MM(2) = 2HEF
MM(3) = 2HGH
C1    = (17.8,-4.0)
C2    = (17.8,-4.0)
```

The constant list must completely satisfy the variable list and there may not be any remaining unused constants.

Dummy variables and variables in blank COMMON cannot be initialized with the DATA statement. Variables in labeled COMMON may be initialized, but only in a BLOCK DATA subprogram.

If a labeled COMMON variable is initialized in more than one program, its value will depend on which program is loaded last. This practice is not recommended.

[†]The size of literal constants is limited to 16 characters in a DATA constant list.

8. PROGRAMS AND SUBPROGRAMS

A complete set of program units executed together as a single job is called an executable program. An executable program consists of one main program and all required subprograms. Subprograms may be defined by the programmer, as described in this section, or may be preprogrammed and contained in the run-time or system libraries.

MAIN PROGRAMS

A main program is comprised of a set of SDS FORTRAN IV-H statements, the first of which (other than comment lines) cannot be one of the following statements, and the last of which is an END statement.

a FUNCTION statement

a SUBROUTINE statement

a BLOCK DATA statement

Main programs may contain any statement except a FUNCTION, SUBROUTINE, ENTRY, or BLOCK DATA statement. Once an executable program has been loaded, execution of the program begins with the first executable statement in the main program.

SUBPROGRAMS

Subprograms are programs which may be called by other programs; they fall into the two broad classes of functions and subroutines.[†] These may be further classified as follows:

Functions

Statement functions

FUNCTION subprograms

Basic external functions

Assembly language functions

Subroutines

SUBROUTINE subroutines

Assembly language subprograms

A function is referenced by the appearance of its identifier within an expression and returns a value (see Chapter 2). Subroutines are referenced with CALL statements and do not necessarily return a value (see Chapter 5). A number of library functions and subroutines are included in SDS FORTRAN IV-H. These are described at the end of this section.

STATEMENT FUNCTIONS

Statement functions are functions that can be defined in a single expression. A statement function definition has the form

$$f(d_1, d_2, d_3, \ldots, d_n) = e$$

where

f	is the name of the function
d_i	is the identifier of a dummy scalar variable (see below)
e	is an arithmetic or logical expression

[†]The BLOCK DATA subprogram, which is neither a function nor a subroutine, is also provided (see Chapter 7).

A statement function must have at least one dummy argument. Statement function dummies are treated only as scalars; they cannot be dummy arrays or subprograms (see "Arguments and Dummies" in this chapter). The expression e should contain at least one reference to each dummy. The identifier f may not appear in the expression, since this would be a recursive definition. References to other statement functions may be made only to previously defined functions.

Examples:

```
F(X) = A * X ** 2 + B * X + C
EI(THETA) = CMPLX(COS(THETA), SIN(THETA))
AVG(PT, NUM, TOT) = 3 *(PT + NUM)/TOT + 1
```

Since each d_i is merely a dummy and does not actually exist, the names of statement function dummies may be the same as the names of other variables in the program. Note, however, that if a statement function dummy is named X, and there is another variable in the program called X, then the appearance of X within the statement function expression refers to the dummy. The only relation between a statement function dummy and any other quantity with the same name is that they will both have the same type. This enables the programmer to declare the types of statement function dummies using explicit (or implicit) type statements.

The statement function itself is typed like any other identifier: it may appear in an explicit type statement; if it does not, it will acquire implicit type (see "Implicit Declarations" in Chapter 7).

A statement function may be referenced only within the program unit in which it is defined. Statement function definitions must precede all executable statements in the program in which they appear.

FUNCTION Subprograms

Functions that cannot be defined in a single statement may be defined as FUNCTION subprograms. These subprograms are introduced by a FUNCTION statement, of the form

FUNCTION $f(d_1, d_2, d_3, \ldots, d_n)$

or

type FUNCTION $f(d_1, d_2, d_3, \ldots, d_n)$

where

f	is the identifier of the function
d_i	is a dummy argument of any of the forms (except asterisk), described in "Arguments and Dummies" later in this chapter
type	is an optional type specification, which may be any of the following:[†]

INTEGER

REAL

DOUBLE PRECISION

COMPLEX

LOGICAL

Every FUNCTION subprogram must have at least one dummy. Values may be assigned to dummies within the FUNCTION subprogram, with certain restrictions (see "Arguments and Dummies").

A FUNCTION subprogram must contain at least one RETURN statement. A RETURN statement should be the last statement in a FUNCTION subprogram; i.e., it should be the last statement executed for each execution of the FUNCTION.

[†]See also "Operational Size Specifications" in Chapter 7.

The identifier of the function must be assigned a value at least once in the subprogram as the argument of a CALL statement, a DO control variable, the variable on the left side of an arithmetic statement, or in an input list (READ statement) within the subprogram.

Within the function the identifier of a FUNCTION subprogram is treated as though it were a scalar variable and should be assigned a value during each execution of the function. The value return for a FUNCTION is the last one assigned to its identifier prior to the execution of a RETURN statement.

A FUNCTION subprogram may contain any FORTRAN statement except a SUBROUTINE statement, another FUNCTION statement, or a BLOCK DATA statement.

FUNCTION statement examples:

```
INTEGER FUNCTION DIFFEQ (R, S, N)
REAL FUNCTION IOU (W, X, Y, Z1, Z2)
FUNCTION EXTRCT (N, A, B, C, V)
LOGICAL FUNCTION VERDAD(E, F, G, H, P)
```

FUNCTION subprogram examples:

```
    COMPLEX FUNCTION GAMMA (Z,N)
    COMPLEX Z
    M = 1
    GAMMA = Z
    DO 5 J = N, 10
    M = M * J
5   GAMMA = GAMMA * (Z + J)
    GAMMA = M * N + Z / GAMMA
    RETURN
    END
```

SUBROUTINE Subprograms

SUBROUTINE subprograms, like FUNCTION subprograms, are self-contained programmed procedures. Unlike FUNCTIONS, however SUBROUTINE subprograms do not have values associated with them and may not be referenced in an expression. Instead, SUBROUTINE subprograms are accessed by CALL statements (see Chapter 5).

SUBROUTINE subprograms begin with a SUBROUTINE statement of the form

SUBROUTINE p($d_1, d_2, d_3, \ldots, d_n$)

or

SUBROUTINE p

where

p is the identifier of the subroutine

d_i is a dummy argument of any of the forms described in "Arguments and Dummies" later in this chapter.

Note that while a FUNCTION must have at least one dummy, a SUBROUTINE need have none.

A SUBROUTINE subprogram must contain at least one RETURN statement; a RETURN statement should be logically the last statement in a SUBROUTINE subprogram (that is, it should be the last statement executed for each execution of the SUBROUTINE).

A SUBROUTINE subprogram may return values to the calling program by assigning values to the d_i or to variables in COMMON storage.

A SUBROUTINE subprogram may contain any FORTRAN statements except a FUNCTION statement, another SUBROUTINE statement, and/or a BLOCK DATA statement. The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. The SUBROUTINE name must not appear in any other statement in the SUBROUTINE program.

ENTRY Statement

The normal entry into a subprogram is at the SUBROUTINE or FUNCTION statement that defines it. Execution begins at the first executable statement following the SUBROUTINE or FUNCTION statement. It is also possible to enter a subprogram at some other point, by using the ENTRY statement, which has the form

ENTRY p

or

ENTRY $p(d_1, d_2, d_3, \ldots, d_n)$

where

p	is the name of the entry point
d_i	is a dummy argument of any of the forms discussed in "Arguments and Dummies", later in this chapter

When control is transferred to a subprogram through an ENTRY statement, execution begins at the first executable statement following the ENTRY statement. The ENTRY statement itself is nonexecutable and does not affect the flow of the program in which it appears. That is, program flow can pass through an ENTRY statement. For example,

```
SUBROUTINE FINISH(N)
END FILE N
ENTRY REWIND(N)
REWIND N
FLAG(N) = 0
RETURN
END
```

The dummy arguments in an ENTRY statement need not agree with those in the FUNCTION on SUBROUTINE statement, nor with those in other ENTRY statements. However, they may agree, if desired. The following statements provide further clarification.

1. When the same dummy name appears in an ENTRY statement and in the FUNCTION or SUBROUTINE statement, it does not refer to two separate entities; it represents the same quantity and, as such, must agree in class and type between the two entry points (e.g., one cannot be real and the other integer).

2. When a dummy name appears at more than one entry point, it need not appear in the same position in the dummy list or correspond to the same argument in the calling program.

3. When a subprogram is entered, all the dummies in the SUBROUTINE, FUNCTION, or ENTRY statement are set up to correspond to the arguments in the call of that statement, thereby overriding any previous correspondence that may have existed. In the example shown above, if the calls

   ```
   CALL FINISH (3)
   CALL REWIND (5)
   ```

   were made, the statement REWIND N would be interpreted as REWIND 3 the first time and as REWIND 5 the second.

Dummy correspondents that are set up by any call on a SUBROUTINE, FUNCTION, or ENTRY statement remain in effect during all subsequent calls on any entry point in the subprogram, unless overridden by the appearance of the same dummy name in a later entry point. Thus, it is permissible to "initialize" a dummy with one call and make use of this initialization on subsequent calls. For example,

```
         ⋮                               SUBROUTINE SUB1(U,V,W,X,Y,Z)
         CALL SUB1 (A,B,C,D,E,F)         RETURN
         ⋮                               ENTRY SUB2(W,T,*)
         CALL SUB2(G,H,&23)              U = V*W + T
         ⋮                               ENTRY SUB3(*)
         CALL SUB3(&14)                  X = W * Y**Z
         ⋮                               IF (U > X) RETURN 1
    23   STOP 'SUB2 ERROR'               RETURN
    14   STOP 'SUB3 ERROR'               END
```

The following actions are taken in the above example:

1. The call on SUB1 initializes U, V, W, X, Y, and Z to correspond to A, B, C, D, E and F, respectively.

2. The call on SUB2 sets T to correspond to H and changes W to correspond to G instead of C. The two assignment statements executed are thus equivalent to

   ```
   A = B*G + H
   D = G * E**F
   ```

   Then, if A is greater than D, return is to statement 23; otherwise, a normal return is taken to the statement following the call on SUB2.

3. The call on SUB3 then changes the alternate return to statement label 14, leaving all other correspondences as they were. Only one assignment statement is executed, which is

   ```
   D = G * E**F
   ```

   Then, if A is greater than D, return is to statement 14; otherwise, a normal return is taken.

4. Note that, if SUB3 were called before SUB2, the action of SUB2 would be unchanged, however, in SUB3 the dummy W would still correspond to C. Then, the assignment statement would have the effect

   ```
   D = C * E**F
   ```

It is an error to reference a dummy argument that has never been initialized. In the above example, if SUB3 were called without having called SUB1, the dummies U, W, X, Y, and Z would not correspond to anything. Care should be taken to avoid situations of this sort.

The following rules also apply to ENTRY statements:

1. The result of a FUNCTION is returned only in the FUNCTION name, never in the ENTRY name. The ENTRY name serves only to identify the location of the entry point and should not be used within the subprogram. In this sense, it is similar to a SUBROUTINE name.

2. ENTRY statements do not alter the rules concerning placement of statement functions. Statement functions may appear after an ENTRY statement only if they still appear before the first executable statement in the subprogram.

3. No subprogram may refer to itself, either directly or indirectly through any of its entry points, nor may it refer to any subprogram whose RETURN statement has not been executed.

4. Like FUNCTION and SUBROUTINE names, ENTRY names are normally available to any program in the executable program.
5. An ENTRY statement may not appear in a main program.
6. Prior to being referenced in an executable statement, every dummy must have appeared in a SUBROUTINE, FUNCTION, or ENTRY statement. In the following example, use of X is correct; use of Y is not.

```
SUBROUTINE ALPHA (X)
X = Y
ENTRY BETA (X,Y)
   .
   .
   .
```

7. If any of the dimensions of an adjustable dummy array are in the dummy list of an ENTRY statement, then the array name must also appear there (see "Adjustables Dimensions" in this section).

ARGUMENTS AND DUMMIES

Dummy arguments provide a means of passing information between a subprogram and the program that called it. Both FUNCTION and SUBROUTINE subprograms may have dummy arguments. A SUBROUTINE need not have any, however, while a FUNCTION must have at least one. Dummies are merely "formal" parameters and are used to indicate the type, number, and sequence of subprogram arguments. A dummy does not actually exist, and no storage is reserved for it; it is only a name used to identify an argument in the calling program. An argument may be any of the following:

a scalar variable

an array element

an array name

an expression

a statement label

a constant of any type (including literal)

a subprogram name

A dummy itself may be classified within the subprogram as one of the following:

a scalar variable

an array

a subprogram

an asterisk denoting a statement label

The chart below indicates the permissible kinds of correspondence between an argument and a dummy.

Argument	Dummy			
	scalar	array	subprogram	asterisk
scalar or array element	yes	yes^{t}	no	no
expression	yes	no	no	no
statement label	no	no	no	yes
array name	yes^{t}	yes	no	no
literal constant	yes^{t}	yes	no	no
subprogram name	no	no	yes	no

^{t}A correspondence of this kind may not be entirely meaningful (see "Dummy Arrays").

A statement label argument is written as

&k

where k is the actual statement label and the ampersand distinguishes the construct as a statement label argument (as opposed to an integer constant).

Within a subprogram, a dummy may be used in much the same way as any other scalar, array, or subprogram identifier with certain restrictions; namely, dummies may not appear in the following types of statements:

COMMON

EQUIVALENCE

DATA

NAME LIST

The reason for the above restriction is that dummies do not actually exist. Furthermore, classification of a dummy as a scalar, an array, or a subprogram identifier occurs in the same manner as with other (actual) identifiers, in both implicit and explicit classifications (see "Classification of Identifiers" in Chapter 7).

In general, dummies must agree in type with the arguments to which they correspond. For example, the following situation is in error because the types of the arguments and the dummies do not agree.

```
COMPLEX C          FUNCTION F (LL, CC)

LOGICAL L          LOGICAL LL

X = F (C, L)       COMPLEX CC
  .                  .
  .                  .
  .                  .
```

Reversing the order of either the arguments in the calling reference or the dummies in the FUNCTION statement would eliminate the error in this example.

There are two exceptions to the rule of type correspondence:

1. A statement number passed as an argument has no type.

2. A SUBROUTINE name (as opposed to a FUNCTION name) has no type.

All arithmetic or logical expressions appearing as actual arguments in the calling program are first evaluated and then placed in a temporary storage location. The address of that temporary storage location is then passed as the argument (this action is referred to as "call by value"). For all other arguments the actual address of the argument is passed (this is referred to as "call by name").

NOTE: All constants are passed by name; therefore, if the called subprogram stores into a dummy corresponding to a constant in the calling sequence, that constant will be changed. Obviously, this is not recommended.

DUMMY SCALARS

Dummy scalars are single valued entities that correspond to a single element in the calling program. Dummies that are not declared (implicitly or explicitly) to be arrays or subprograms are treated as scalars.

DUMMY ARRAYS

A dummy argument may be defined as an array, by the presence of its identifier in any array declaration within the subprogram (the fact that a calling argument is an array does not in itself define the corresponding dummy to be an array). A dummy array does not actually occupy any storage, it merely identifies an area in the calling program. The subprogram assumes that the argument supplied in the calling statement defines the first (or base) element of an actual array and calculates subscripts from that location.

Normally, a dummy array is given the same dimensions as the argument array to which it corresponds. This is not necessary, however, and useful operations can often be performed by making them different. For example,

```
DIMENSION A(10, 10)        SUBROUTINE OUT (B)
CALL OUT (A(1,6))          DIMENSION B(50)
   .                          .
   .                          .
   .                          .
```

In this case, the 1-dimensional dummy array B corresponds to the last half of the 2-dimensional array A (i.e., elements A(1,6) through A(10, 10)). However, since an array name used without subscripts as an argument refers to the first element of the array, if the calling statement were

```
CALL OUT(A)
```

the dummy array B would correspond to the first half of the array A.

Arguments that are literal constants are normally received by dummy arrays. A literal constant is stored as a consecutive string of characters in memory, and its starting location is passed as the argument address. For instance, in the example

```
CALL FOR('PHILIP MORRIS')        SUBROUTINE FOR(M)
   .                             DIMENSION M (5)
   .
   .
```

the following correspondences hold:

```
M(1) = 4HPHIL
M(2) = 4HIPƀM
M(3) = 4HORRI
M(4) = 4HSƀƀƀ
M(5)   is undefined and should not be referenced
```

where ƀ represents the character blank. Literal constants are filled out with trailing blanks to the nearest word boundary (multiple of four characters). Therefore, passing such a constant to a dummy of a type that occupies more than one word per element[†] (e.g., double precision) may result in dummy elements that are only partially defined. For this reason, integer arrays are recommended.

If an array corresponds to something that is not an array or a literal constant, the latter will correspond to the first element of the array. This is true whether the calling argument is an array and the dummy is not, or vice versa. For example, if the calling argument is a scalar and the dummy is an array, references in the subprogram to elements of the array other than the first element will correspond to whatever happens to be stored near the scalar. Care must be taken in creating correspondences of this nature since they may depend upon a particular implementation.

ADJUSTABLE DIMENSIONS

Since a dummy array does not actually occupy any storage, its dimensions are used only to locate its elements, not to allocate storage for them. Therefore, the dimensions of a dummy array do not have to be defined within the subprogram in the normal manner. Instead, any or all the dimensions of a dummy array may be specified by dummy scalar variables rather than by constants. This permits the calling program to supply the dimensions of the dummy array each time the subprogram is called. The following statements demonstrate adjustable dimensions:

```
DIMENSION P(10,5), Q(9,3)        FUNCTION SUM (R, N, M)
X = SUM(P, 10, 5)                DIMENSION R(N, M)
Y = SUM(Q, 9, 3)
   .
   .
   .
```

[†]See "Allocation of Variable Types" in Chapter 7.

Only a dummy array can be given adjustable dimensions, and the dimensions must be specified by dummy integer scalars. The variables used as adjustable dimensions may be referenced elsewhere in the subprogram but should not be changed. In particular, the appearance of any of these variables in the dummy list of a succeeding ENTRY statement constitutes a change. Therefore, an ENTRY statement may not contain any dimensions of an adjustable array itself. The appearance of the array name in the ENTRY statement causes the array bounds to be recomputed (see "ENTRY Statement").

DUMMY SUBPROGRAMS

A dummy subprogram must correspond to an argument that is a subprogram name, and it is the only kind of dummy that can do so. The dummy name merely serves to identify a closed subprogram whose actual location is defined by the calling program. Therefore, a call on a dummy subprogram is actually a call on the subprogram whose name is specified as the argument. A dummy subprogram is classified in the same manner as any other subprogram (see "Classification of Identifiers" in Chapter 7).

Example:

```
    EXTERNAL SIN, DSIN, SQRT, DSQRT        FUNCTION DIFF(F, DF, Z)

    A = DIFF(SIN, DSIN, X)                 DOUBLE PRECISION DF

    B = DIFF(SQRT, DSQRT, Y)               DIFF = DABS(F(Z) - DF(DBLE(Z)))

    .                                      RETURN
    .
    .                                      END
```

A subprogram identifier, to be passed as an argument, must previously appear in an EXTERNAL statement (otherwise, it may be classified as a scalar variable).

LIBRARY SUBPROGRAMS

SDS FORTRAN IV-H includes a number of library subprograms. These are specially recognized by the compiler, which generates special machine codes for them. Most of the library subprograms are functions, although several utility subroutines are also provided

BASIC EXTERNAL FUNCTIONS

The basic external function subprograms evaluate commonly used mathematical functions. These subprograms have a special type that is known to the compiler. This type is not necessarily the same as the type it would acquire by implicit typing rules. The arguments to these functions must have the proper type, as shown in Table 8.

Table 8 lists the function subprograms provided by SDS FORTRAN IV-H. When a formula is shown in the column "Definition of Function", it is not necessarily the formula that is actually used in implementing the function; it is intended only to clarify the definition of function.

ADDITIONAL LIBRARY SUBPROGRAMS

In addition to the functions listed in Table 8, the following subprograms are supplied in the SDS FORTRAN IV-H library:

<u>EXIT</u>

Form:

```
    CALL EXIT
```

EXIT is typed on the listing output device and the job is terminated.

Table 8. Basic External Functions

Function Name	Number of Arguments	Type of Argument	Type of Result	Definition of Function
ABS	1	Real*4	Real*4	Absolute value
AIMAG	1	Complex*8	Real*4	Imaginary part of argument expressed as a real value.
AINT	1	Real*4	Real*4	Integer part of argument expressed as a real value.
ALOG	1	Real*4	Real*4	Natural logarithm (base e).
ALOG10	1	Real*4	Real*4	Common logarithm (base 10).
AMAX0	$N \geq 2$	Integer*4	Real*4	Maximum value for integer values.
AMAX1	$N \geq 2$	Real*4	Real*4	Maximum value for real values.
AMIN0	$N \geq 2$	Integer*4	Real*4	Minimum value for integer values.
AMIN1	$N \geq 2$	Real*4	Real*4	Minimum value for real values.
AMOD	2	Real*4	Real*4	Arg_1 (mod arg_2). Evaluated as $\text{arg}_1 - \text{arg}_2 * \text{AINT}(\text{arg}_1/\text{arg}_2)$ (i.e., the sign is the same as arg_1) Function undefined if $\text{arg}_2 = 0$.
ATAN ATAN2	1 2	Real*4 Real*4	Real*4 Real*4	Arctangent in radians. Arg_1 = ordinate (y), arg_2 = abscissa(x). If arg_2 not present, 1 is assumed. Result (R) is arctangent of $\text{arg}_1/\text{arg}_2$ quadrant allocated in the range $-\pi < R \leq \pi$; ATAN(0, 0) = 0.
CABS	1	Complex*8	Real*4	Complex absolute value (i.e., modulus). $\text{CABS}(x+iy) = \sqrt{x^2+y^2}$
CCOS	1	Complex*8	Complex*8	Complex cosine. $\text{CCOS}(Z) = (e^{iZ}+e^{-iZ})/2.$
CDABS	1	Complex*16	Real*8	Double-complex absolute value (modulus). See CABS.
CDCOS	1	Complex*16	Complex*16	Double-complex cosine. See CCOS.
CDEXP	1	Complex*16	Complex*16	Double-complex exponential. See CEXP.
CDLOG	1	Complex*16	Complex*16	Double-complex natural logarithm (base e). See CLOG.
CDSIN	1	Complex*16	Complex*16	Double-complex sine. See CSIN.
CDSQRT	1	Complex*16	Complex*16	Double-complex square root. See CSQRT.
CEXP	1	Complex*8	Complex*8	Complex exponential (e**arg). CEXP(x+iy) = EXP(x) (COS(y) + i SIN(y)).
CLOG	1	Complex*8	Complex*8	Complex natural logarithm (base e). CLOG (Z) = CLOG(x+iy) = u + iv = ln Z + i ATAN(y, x), allocated such that $-\pi < v \leq \pi$.
CMPLX	2	Real*4	Complex*8	Converts two non-complex numbers to a complex number. CMPLX(x, y) = x + iy.
CONJG	1	Complex*8	Complex*8	Complex conjugate. CONJG(x+iy) = x - iy (has no effect if arg not complex).
COS	1	Real*4	Real*4	Cosine of angle in radians.

Table 8. Basic External Functions (cont.)

Function Name	Number of Arguments	Type of Argument	Type of Result	Definition of Function
CSIN	1	Complex*8	Complex*8	Complex sine. CSIN(Z) = $(e^{iZ} - e^{-iZ})/(2j)$
CSQRT	1	Complex*8	Complex*8	Complex square root. CSQRT(Z) = u + iv = $e^{(\ln Z)/2}$, allocated such that $u \geq 0$.
DABS	1	Real8*	Real*8	Double-precision absolute value.
DATAN	1	Real*8	Real*8	Double-precision arctangent in radians. See ATAN.
DATAN2	2	Real*8	Real*8	
DBLE	1	Real*4	Real*8	Argument converted to double precision.
DCMPLX	2	Real*8	Complex*16	Converts two non-complex numbers to a double-complex number. See CMPLX.
DCONJG	1	Complex*16	Complex*16	Double-complex conjugate. See CONJG.
DCOS	1	Real*8	Real*8	Double-precision cosine of angle in radians.
DEXP	1	Real*8	Real*8	Double-precision exponential (e**arg).
DFLOAT	1	Integer*4	Real*8	Argument converted to double precision. Same as DBLE, but used with integer arguments.
DIM	2	Real*4	Real*4	Positive difference. DIM(x, y) = max(x-y, 0)
DLOG	1	Real*8	Real*8	Double-precision natural logarithm (base e).
DLOG10	1	Real*8	Real*8	Double-precision common logarithm (base 10).
DMAX1	$N \geq 2$	Real*8	Real*8	Double-precision maximum value.
DMIN1	$N \geq 2$	Real*8	Real*8	Double-precision minimum value.
DMOD	2	Real*8	Real*8	Double-precision arg_1 (mod arg_2). See AMOD.
DSIGN	2	Real*8	Real*8	Double-precision magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
DSIN	1	Real*8	Real*8	Double-precision sine of angle in radians.
DSQRT	1	Real*8	Real*8	Double-precision square root (positive value).
DTANH	1	Real*8	Real*8	Double-precision hyperbolic tangent.
EXP	1	Real*4	Real*4	Exponential (e**arg).
FLOAT	1	Integer*4	Real*4	Argument converted to a real value.
IABS	1	Integer*4	Integer*4	Integer absolute value.
IDIM	2	Integer*4	Integer*4	Integer positive difference. IDIM(j, k) = j - MIN(j, k).
INT	1	Real*4	Integer*4	Argument converted to an integer value.
IFIX	1	Real*4	Integer*4	
IDINT	1	Real*8	Integer*4	
ISIGN	2	Integer*4	Integer*4	Integer magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
MAX0	$N \geq 2$	Integer*4	Integer*4	Integer maximum value.

Table 8. Basic External Functions (cont.)

Function Name	Number of Arguments	Type of Argument	Type of result	Definition of Function
MAX1	$N \geq 2$	Real*4	Integer*4	Integer maximum value.
MIN0	$N \geq 2$	Integer*4	Integer*4	Integer minimum value.
MIN1	$N \geq 2$	Real*4	Integer*4	Integer minimum value.
MOD	2	Integer*4	Integer*4	Arg_1 (mod arg_2). Evaluated as $arg_1 - arg_2 * [arg_1/arg_2]$ where the brackets indicate integer part; i.e., the sign is the same as arg_1. Function is undefined if $arg_2 = 0$.
REAL	1	Complex*8	Real*4	Real part of a complex number.
SIGN	2	Real*4	Real*4	Magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
SIN	1	Real*4	Real*4	Sine of angle in radians.
SNGL	1	Real*8	Real*4	Argument converted to a value with real (single) precision.
SQRT	1	Real*4	Real*4	Square root (positive value).
TANH	1	Real*4	Real*4	Hyperbolic tangent. For complex, $TANH(Z) = SINH(Z)/COSH(Z) = (e^Z - e^{-Z})/(e^Z + e^{-Z})$

SLITET – Sense Light Test

Form:

```
CALL SLITET (n,v)
```

where

n is an integer expression specifying which sense light is to be tested ($1 \leq n \leq 4$)

v is an integer variable in which the result of the test will be stored

Sense light n is tested. If the sense light is on, the value 1 will be stored in v; if it is off, the value 2 will be stored. Following the test, the sense light will be turned off.

SLITE – Set Sense Light

Form:

```
CALL SLITE (n)
```

where

n is an integer expression ($0 \leq n \leq 4$)

If n is 0, all sense lights will be turned off; if n is 1, 2, 3, or 4, the corresponding sense light will be turned on.

OVERFL – Floating Overflow Test

Form:

```
CALL OVERFL (s)
```

where

s is an integer variable into which will be stored the result of the test

If a floating overflow has occurred, s is set to 1; if no overflow condition exists, s is set to 2. If a floating underflow condition exists, s is set to 3. The machine is left in a no overflow (underflow) condition following the test. Overflow and underflow are defined in the Sigma computer reference manual.

DVCHK – Divide Check

Form:

CALL DVCHK (s)

where

s is an integer variable into which will be stored the result of the test

This is another entry to the OVERFL subprogram described above.

DUMP

A call to the DUMP subprogram has the form

CALL DUMP(A_1, B_1, F_1, ..., A_n, B_n, F_n)

where

A and B are variable data names that indicate the limits of storage to be dumped; either A or B may represent upper or lower limits. The arguments A and B should be in the same program (main program or subprogram) or same COMMON block.

F_i is an integer indicating the dump format desired:

F_i = 0 Hexadecimal
2 Logical*4
4 Integer*4
5 Real*4
6 Real*8
7 Complex*8
8 Complex*16
9 Literal

any other value of F_i is illegal.

If the argument F_n is omitted, it is assumed to be zero, and the dump will be in hexadecimal format.

A call to this subroutine causes the indicated limits of storage to be dumped and execution to be terminated.

PDUMP

A call to the PDUMP subprogram has the form

CALL PDUMP (A_1, B_1, F_1, ..., A_n, B_n, F_n)

where

A, B, and F are the same as for DUMP

This call causes the indicated limits of storage to be dumped and execution to be continued.

APPENDIX A. SDS SIGMA FORTRAN IV-H CHARACTER SETS

The standard character set for use with SDS Sigma FORTRAN IV and FORTRAN IV-H is the EBCDIC (Extended Binary-Coded-Decimal Interchange Code). This character set is illustrated in Table 9.

Table 9. SDS EBCDIC
(Extended Binary-Coded-Decimal Interchange Code)

Character	EBCDIC Card Code	9-Channel Magnetic Tape Hexadecimal Code	Character	EBCDIC Card Code	9-Channel Magnetic Tape Hexadecimal Code
A	12-1	C1	Blank	Blank	40
B	12-2	C2	¢ (a)	12-2-8	4A
C	12-3	C3	.	12-3-8	4B
D	12-4	C4	<	12-4-8	4C
E	12-5	C5	(	12-5-8	4D
F	12-6	C6	+	12-6-8	4E
G	12-7	C7	|	12-7-8	4F
H	12-8	C8	&	12	50
I	12-9	C9			
J	11-1	D1	! (a)	11-2-8	5A
K	11-2	D2	$	11-3-8	5B
L	11-3	C3	*	11-4-8	5C
M	11-4	D4	)	11-5-8	5D
N	11-5	D5	;	11-6-8	5E
O	11-6	D6	¬ (a)	11-7-8	5F
P	11-7	D7	-	11	60
Q	11-8	D8			
R	11-9	D9	/	0-1	61
S	0-2	E2	,	0-3-8	6B
T	0-3	E3	%	0-4-8	6C
U	0-4	E4	- (a)	0-5-8	6D
V	0-5	E5	>	0-6-8	6E
W	0-6	E6	? (a)	0-7-8	6F
X	0-7	E7			
Y	0-8	E8	:	2-8	7A
Z	0-9	E9	#	3-8	7B
0	0	F0	@	4-8	7C
1	1	F1	'	5-8	7D
2	2	F2	=	6-8	7E
3	3	F3	" (a)	7-8	7F
4	4	F4			
5	5	F5			
6	6	F6			
7	7	F7			
8	8	F8			
9	9	F9			

(a) This character is not included in the SDS Standard 56-graphic character set, used by some line printers.

The SDS Sigma internal hexadecimal representation (in memory) of every character is the same as the magnetic tape representation. However, it is not advisable for the FORTRAN programmer to take advantage of these numeric representations since this tends to make the program machine-dependent.

APPENDIX B. SDS SIGMA FORTRAN IV-H STATEMENTS

Statement	Executable	Nonexecutable	Page
ASSIGN	X		18
Assignment	X		15
BACKSPACE	X		46
BLOCK DATA		X	58
CALL	X		20
COMMON		X	52
COMPLEX		X	50
CONTINUE	X		24
DATA		X	59
DIMENSION		X	49
DO	X		21
DOUBLE PRECISION		X	50
END		X	25
END FILE	X		46
ENTRY		X	64
EQUIVALENCE		X	55
EXTERNAL		X	58
FORMAT		X	33
FUNCTION		X	62
GO TO	X		17
IF	X		19
IMPLICIT		X	49
INTEGER		X	50
LOGICAL		X	50
NAMELIST		X	30
PAUSE	X		24
PRINT	X		29
PUNCH	X		29
READ	X		29
REAL		X	50
RETURN	X		21
REWIND	X		46
STOP	X		25
SUBROUTINE		X	63
Statement Function Definition		X	61
WRITE	X		1, 28, 29, 30

INDEX