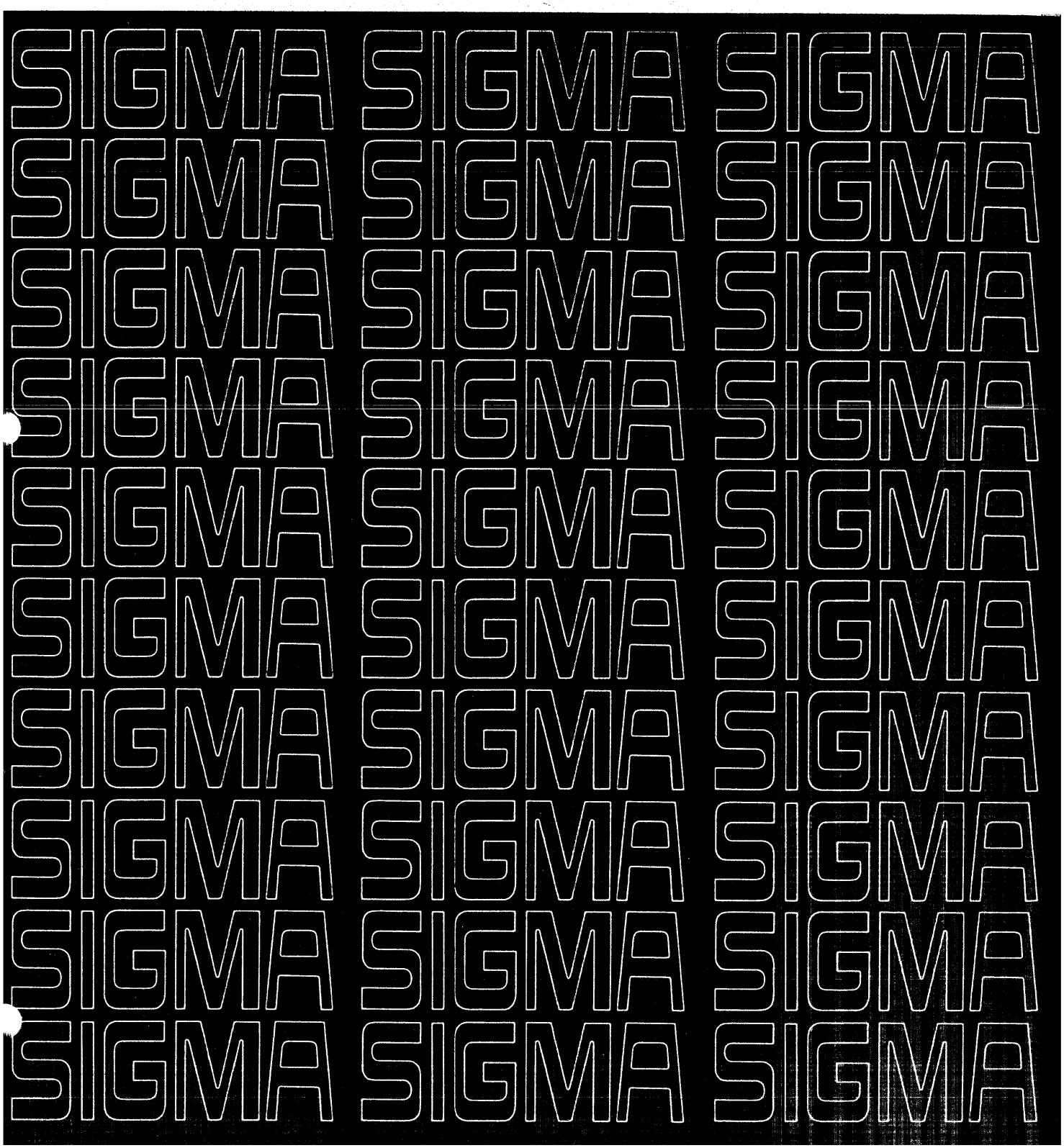
SDS

Scientific Data Systems
A XEROX COMPANY

XDS SIGMA 5/7 FLAG

Reference Manual



Price: $4.00

FLAG
REFERENCE MANUAL

for

XDS SIGMA 5/7 COMPUTERS

PRELIMINARY EDITION

90 16 54A

September 1969

XDS

Xerox Data Systems/701 South Aviation Boulevard/El Segundo, California 90245

Printed in U.S.A.

ACKNOWLEDGMENT

The FLAG (Fortran Load And Go) Compiler was developed by Norman Wheeler, Robert Horton, Phillip Sherrod, and Douglass Henry under the direction of Dr. George Haynam of the Vanderbilt University Computer Center, Nashville, Tennessee.

In cooperation with Vanderbilt University, Xerox Data Systems is making the FLAG Compiler available to those customers who require the characteristics of a fast load-and-go FORTRAN.

The reference material in this manual was largely taken from the XDS Sigma 5/7 FORTRAN IV-H Reference Manual (XDS 90 09 66) though several important sections have been excerpted from the XDS Sigma 5/7 FORTRAN IV Reference Manual (XDS 90 09 56), notably all the material in Chapter 6 dealing with FORMAT statements and specifications and with extended input/output capabilities. Furthermore, FLAG provides most of the function subprograms that are available to FORTRAN IV users. These subprograms are listed in "Table 9 Intrinsic and Basic External Functions", taken from the previously mentioned FORTRAN IV Reference Manual.

RELATED PUBLICATIONS

Title	Publication No.
XDS Sigma 5 Computer Reference Manual	90 09 59
XDS Sigma 7 Computer Reference Manual	90 09 50
XDS Sigma 5/7 Batch Processing Monitor Reference Manual	90 09 54
XDS Sigma 5/7 Batch Processing Monitor Operations Manual	90 11 98
XDS Sigma 5/7 FORTRAN IV-H Reference Manual	90 09 66
XDS Sigma 5/7 FORTRAN IV Reference Manual	90 09 56
XDS Sigma 5/7 FORTRAN IV Operations Manual	90 11 43
XDS Sigma 5/7 FORTRAN IV Library Technical Manual	90 15 24

CONTENTS

APPENDIXES

ILLUSTRATIONS

TABLES

1. INTRODUCTION

How This Manual is Organized

Most of the material in this manual was taken from the Sigma 5/7 FORTRAN IV-H Reference Manual (XDS 90 09 66B), though several important sections are from the Sigma 5/7 FORTRAN IV Reference Manual (XDS 90 09 56C). For the convenience of those who may already be familiar with XDS FORTRAN IV-H, material in this manual that differs from the FORTRAN IV-H Manual is indicated by a bracket in the left hand margin of the page.

The remainder of this Chapter summarizes the most important features of FLAG and then briefly presents information of general interest to the new user. Chapters 2 through 8 are a detailed description of the FLAG language. Chapter 9 contains the essential operations information for compiling and running FLAG programs.

Users already familiar with XDS FORTRAN IV-H will probably perfer to scan Chapters 1 through 8 noting the changed areas marked by brackets, and then read Chapter 9 for an explanation of FLAG operations. Such users will thus be able to start running FLAG programs with minimum delay.

The FLAG Compiler

The FLAG (FORTRAN Load And Go) system for XDS Sigma 5/7 computers is essentially a FORTRAN IV-H compiler designed to be compatible with other compilers of this class. However, FLAG provides to the user a unique set of operating characteristics:

- Comprehensive diagnostic messages at compile and execute time.
- Fast compilation.
- Significant reduction in total processing time for small-to-medium sized programs.
- Special accounting and processing features to minimize Monitor system overhead.

FLAG may be used in preference to standard FORTRAN compilers when the user is in the debugging phase of developing his program. Further, it should be the primary FORTRAN compiler system in the typical university environment where the job stream contains numerous small programs, many of which are written by novice programmers.

FLAG Programs

FLAG 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[†], that perform computation, input/output operations, and program flow control.
2. Nonexecutable statements[†], 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 a FLAG 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. Chapter 5, "Control Statements", contains a more extensive discussion of statement labels.

[†] See Appendix A.

The body of each statement is written in columns 7 through 72, but if additional space is required, a statement may be continued. FLAG 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 FLAG is

Letters[†]:	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)[†]	+ - * / = ( ) . , $ ' & blank
Special characters: (other)	< > ; : @ # % ¢ ? ! – | _ "

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

Figure 1 illustrates a sample FLAG program. An explanation is given in Table 1.

PROBLEM Sample

PROGRAMMER

SDS
SCIENTIFIC DATA SYSTEMS

FORTRAN CODING FORM

Identification FACTOR

C FOR COMMENT

STATEMENT NUMBER	Cont.	FORTRAN STATEMENT
		C ROUTINE TO CALCULATE FACTORIALS
		C *****
		INTEGER FACTOR, K
		K = 1
		READ (1,5) FACTOR
10		IF (FACTOR) 12,13,11
11		K = K * FACTOR
		FACTOR =
		C FACTOR - 1
		GO TO 10
12		K = 0
13		WRITE (108,6) K
		STOP
5		FORMAT (I6)
6		FORMAT ('K=',I20)
		END

Figure 1. Sample XDS FLAG Program

[†]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.

Table 1. Sample Program

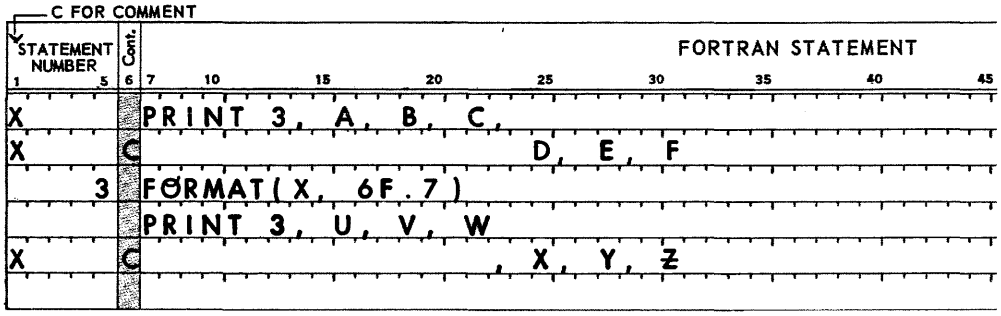
Line	Meaning
1, 2	The character C in column 1 defines these lines as comments.
3	A nonexecutable statement that defines to the compiler the variables FACTOR and K as integers.
4	An assignment statement that sets K equal to 1.
5	An input command that causes the value of FACTOR to be read into storage. 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 < 0, control is transferred to statement 12. If FACTOR = 0, control is transferred to statement 13. If FACTOR > 0, 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.
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.

Conditional Compilation – X Cards

FLAG provides a means for conditional compilation of statements. Any line that contains an X in column 1 is processed as a statement only when the CX option is specified on the FLAG control card (see Chapter 9). Otherwise, the card is treated as a comment.

This feature enables the programmer to include in his program additional statements for checkout purposes, such as intermediate output and special error checking. When checkout has been completed, these statements do not have to be removed from the program. Instead, the program is compiled without the CX option and the statements are treated as comments only. These statements remain in the listing, however, and may serve as documentation or checkout procedures. Also, they may easily be reinstated at any time.

Continuation lines for X cards should also be X cards; furthermore, a normal line may have a continuation line that is also an X card. For example:



2. DATA

Numerical quantities – constants and variables – as distinguished in FLAG 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 a FLAG 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 Chapter 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 3.141592653589793238 46

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 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, used to name variables as well as subprograms and COMMON blocks. (See Chapters 7 and 8 for discussions of COMMON and subprograms.)

Identifiers in FLAG 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 FLAG; however, for clarity, it is advisable not to use identifiers that correspond to FLAG 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. FLAG 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 may be any expression that has a resultant mode of integer, real, or double precision; if the result is not integer it is converted to integer mode (by truncation).

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. 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 FLAG. In addition, the user may define his own functions (see 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 exponentiations are performed left to right. 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

During evaluation of mixed expressions the mode of an operand will be converted, if necessary, so that the resultant mode of each operation will be as shown in Table 2.

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

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

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 recognized by the FLAG compiler. 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 is equal to e, but rather it sets 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:

Symbol	Meaning
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 FLAG 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. FLAG 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 $1 \leq j \leq n$. 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 (a1, a2, a3,...,an)
```

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 or logical 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 determinant 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 should not appear in a main program.

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 entry to the subprogram did not contain any asterisks in the dummy list, the RETURN statement will cause a compile-time diagnostic to be produced.

Examples:

Calling Program

```
33  CALL IT (LOCK, RET, QR, &11, &883)
    .
    .
    .
66  X(8) = Y(C, K) + CHEBY(Z, Y)
    .
    .
    .
```

Subprograms

```
SUBROUTINE IT (i, X, P, *, *)
.
.
.
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.

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.

Caution: If the entire number of iterations specified for a DO loop is executed, the value of v becomes undefined when program control passes out of the DO loop.

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. Up to 25 DO ranges may 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 ($1 \leq n \leq 99999$).

'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 FLAG compiler that it has reached the physical end of a program. The statement must appear in the form

END

If program control reaches an END statement during the execution of any program (or subprogram) the effect is that of a STOP statement.

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 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 statements, will be processed as though they were END statements.

Processed as END Statements

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

Not Processed as END Statements

```
        6   7.....
            END FILE
        1   2
            END RA
        X               TE = A * B
            END (I, J
        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

   $$B(1,1), B(2,1), B(3,1), \ldots, B(1,2), B(2,2), \ldots, B(j,k)$$

   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 parentheses 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 FLAG. There are no standard unit assignments for magnetic tapes or random access devices.

Table 6. Standard Unit Assignments

Unit Number	Standard Assignments
5,105	Card reader
6,108	Line printer
7,106	Card punch

If nonstandard unit numbers are used in a program they must be assigned to the desired device by use of ASSIGN control cards placed in front of the FLAG control card (see Chapter 9). The default function of an assigned, non-standard unit number is OUTIN (scratch mode). If a unit is to be assigned to an input device (e.g., card

reader or magnetic tape) the IN option should be specified on the ASSIGN control card. Some sample assignments follow. To assign unit number 1 to the card reader:

```
!ASSIGN F:1, (DEVICE, SI), (IN)
```

To assign unit number 2 to the line printer:

```
!ASSIGN F:2, (DEVICE, LO, L)
```

To assign unit number 3 to a labeled magnetic tape with serial number 'PHS' from which data will be read:

```
!ASSIGN F:3, (LABEL, DATAFILE), (INSN, PHS), (IN)
```

To assign unit number 4 to a scratch disc file:

```
!ASSIGN F:4, (FILE, TEMP4)
```

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 comma may optionally precede the list k.

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 XDS 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 FLAG. 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	FLAG 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 to be output on device 106.

PRINT Statement

The form of the FORTRAN II 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 to be output on device 108 (see also "Carriage Control for Printer Output" in this Chapter).

"FORMAT-Free" READ and PRINT Statements

"FORMAT-free" forms of the READ and PRINT statements are also provided. The general forms are as follows:

READ, k

PRINT, k

where k is an input/output variable list of the usual form. Output values will be printed 8 per line; input values for "FORMAT-free" READ should be separated by either a comma or one or more blanks; if more than 8 values are to be read by one READ statement the values should be punched 8 per card using as many cards as necessary. (The actual FORMAT specifications used by "FORMAT-free" READ and PRINT are (8G) and (8*(2XG.6)) respectively).

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 Statements

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 Statement

The NAME LIST statement is used to define the variables that may be processed by INPUT statements. It has the form

NAME LIST $v_1, v_2, v_3, \ldots, v_n$

where the v_i are scalar or array identifiers. Dummy variables may not appear.

When an array name appears in a NAME LIST declaration, all elements of the array may be processed by an INPUT statement.

A NAME LIST statement with no identifiers following it causes all appropriate variables that appear in the program to be placed in the name list; i.e., all nondummy scalars and arrays. This can be helpful during program checkout, since it enables the user to input any variable without knowing at compile time which variables it will be desirable to input.

The NAME LIST variables defined in a program unit are independent of those defined in any other program units. Each program has its own NAME LIST. This means the following:

1. A variable that appears in one program may not be processed by an INPUT statement in another program unless, for example, the variable is in COMMON and also appears in the other program.

2. If two or more programs have separate variables with the same name, it is possible to input into either of them. It simply depends on which program is doing the inputting. There is no conflict between the separate NAME LISTs.

Examples of NAME LIST statements:

```
NAME LIST T, G, I, F, RATE, COUNT, ITEM QUANTITY

NAME LIST NORM, BOB, PHIL, DOUG

NAME LIST
```

Simplified Input/Output

This is the most straightforward form of input/output. It does not require the programmer to learn anything about FORMAT statements or other kinds of input/output which, although more versatile, are also more complicated. It can process every type of variable and is suitable for almost any FORTRAN application that does not require special editing or formatting. Values may be written out in a natural form chosen by the compiler. Similarly, data can be input in a very free form without the usual FORTRAN requirement that the user know exactly what FORMAT is controlling the input operation.

OUTPUT Statement

The OUTPUT statement may have any of the following forms:

```
OUTPUT k
OUTPUT, k
OUTPUT(u) k
OUTPUT(u), k
```

where

k is an output list, consisting of variables, expressions, and/or character strings.

u defines the logical unit number of the device on which the output data is to be written. It may be any unsigned integer or integer variable. If no unit number is specified, the output will automatically be on unit 108, which is the standard print unit.

The name of each list item will be output, followed by an equal sign and then the value of the item, in an appropriate format.

For example,

```
OUTPUT X, Y, X + Y, SQRT(X**2+Y**2)
```

might produce the following lines of output:

```
X = .500000
Y = 1.20000
X + Y = 1.70000
SQRT(X**2+Y**2) = 1.30000
```

Complex values are output as complex constants; the other data types are also output in natural forms, as shown in the following example:

```
DOUBLE PRECISION D
COMPLEX C
LOGICAL L
OUTPUT(6), I, R, D, C, L
```

which might result in these lines:

```
I = 79
R = 4370.72
D = 99301.3922310385
C = (-56.2234,334.882)
L = T
```

Each value is written on a separate line, beginning in column 2 (so that no carriage control will take place). The maximum width of any line is 132 characters; excess characters will be lost.

If the first item in the output list begins with a left parenthesis, and no unit number is specified, there must be a comma after the word OUTPUT. Otherwise, the list item will appear to be a unit number. For example,

```
OUTPUT, (A + B), X(I)
```

When an unsubscripted array name appears in the output list, the entire array is output in storage order (see "Array Storage", Chapter 7). For example,

```
DIMENSION M(2, 2), A(2)
OUTPUT M, A
```

which could print the following:

```
M =  553
      -4
       0
     11245
A =  .472962
     -33.0000
```

To output headings or other alphanumeric information, a list item may be a character string enclosed in quotes (i.e., a literal constant). In this case, no equal sign or value will be generated; only the character string itself is output, as in

```
OUTPUT 'FINAL COORDINATES'
```

which generates the line

```
FINAL COORDINATES
```

Implied DO loops may be used, as in the following example. Note that the list begins with a parenthesis and is therefore preceded by a comma.

```
OUTPUT, (K, A(K)/B(K), K = N1, N2)
```

which could produce this output:

```
K = 3
A(K)/B(K) = 14.6135
K = 4
A(K)/B(K) = 15.0873
etc.
```

Another feature is provided which enables OUTPUT records to be read subsequently by a "NAMELIST" INPUT statement. An OUTPUT list item may consist of a single asterisk (*), which will cause the characters *END* to be output on a record. This will cause an INPUT statement to terminate reading. For example,

```
OUTPUT (4) X, I, J, AA, *
```

The actual format specifications used to output the various types of data are shown below, although the programmer need not be concerned about them, since they are provided automatically. Note that all the formats are widthless.

Data Type	Format Specifications
integer	I
real	1PG.6
double precision	1PG.15
complex	1P, 1H(G.6, -X, 1H, G.6, -X, 1H)
double complex	1P, 1H(G.15, -X, 1H, G.15, -X, 1H)
logical	L

INPUT Statement Using NAME LIST

The INPUT statement is the counterpart of the OUTPUT statement, except that it is written without a specific list of variables. The forms of the statement are shown below.

```
INPUT
INPUT(u)
```

where u is the unit number, as described under the OUTPUT statement. When not specified, it is assumed to be 105.

This form of the INPUT statement is designed to do self-identified input. That is, the variables being input are identified by the input itself, rather than being named explicitly in an input list within the program. This enables the user to decide at run time which variables (if any) are to be input, and to select different input variables from run to run.

This statement processes records of the form

$$r_1;r_2;r_3;\ \ldots\ ;r_n$$

where the r_i are each a form of replacement (similar to an assignment statement). Either semicolons (;), as indicated, or commas may be used to separate the r_i. There may be any number of r_i on a record. Except within constants, blanks are ignored.

Each of the replacements r_i may take one of the following forms:

$$v = c$$

$$a(s_1, s_2, s_3, \ldots, s_n) = c$$

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

In the first form,

- v is a scalar variable and
- c is a constant of an appropriate type.

The second form is used to input into an array element. Here,

- a is the identifier of an array,
- s_i is a subscript, which may be a signed or unsigned integer, and
- c is an appropriate constant.

The third form specifies input into an entire array, in the same manner as described for explicit input/output lists (see "Input/Output Lists"). In this case,

- a is an array name,
- c_j is a constant of an appropriate type, and
- m is equal to the number of elements in the array.

When an entire array is specified as above, the constants c_j are assigned to successive array elements in the order in which the array is stored (i.e., columnwise; see Figure 2, Chapter 7). Note that there must be the same number of constants as there are elements in the array. The constants may appear on separate input records, in which case the separating comma is optional.

Example:

```
ALPHA = 1.7302, -67,
        4E-5
        .87, 24.0983281640957
100000000000.  ;  X = 0; etc.
```

This is the only case in which a single INPUT item (an r_i) may overlap from one record to another.

Variables that are to be processed by an INPUT statement must have been referenced by a NAME LIST statement so that their names can be recognized at run time. Note that this can be done by using a NAME LIST statement with no identifier list after it, which causes all permissible scalars and arrays to be placed in the name list.

The permissible forms of a constant that appears on an input record depend on the type of the variable to which the constant is to be assigned.

1. For integer, real, and double precision variables, the constant is scanned according to a G.0 (widthless) field specification. The constant may therefore take any of the forms described under "Numeric Input Strings" for widthless formats. This includes all the forms discussed in Chapter 2 for integer, real, and double precision constants (except Hollerith). Note, however, that since they are scanned with a widthless format, these constants are terminated at the first blank that follows a digit or decimal point. A good rule is not to embed blanks within constants. If this rule is followed, INPUT replacements may be written exactly as assignment statements.

2. For complex variables, the constant must be of the form

   (c_1, c_2)

   where c_1 and c_2 may each be any of the forms discussed above for real data. The meaning is the same as for a complex constant as described in Chapter 2.

3. For logical variables, the constant may take either of the forms

   .TRUE. or .FALSE.

   or the constant may be any character string that can be processed by a widthless format; that is, one in which the letter T or F appears. Such a field is terminated by the first comma or nonleading blank.

   Thus, T, F, TRUE, and FALSE are all permissible input strings for logical variables.

Note that if the items in an OUTPUT statement are restricted to scalars, array elements (with constant subscripts), and arrays, the resultant output records can be processed by an INPUT statement.

INPUT processing terminates when an asterisk character is found in the input string, wherever that may be. For example,

```
T = 55E-2; A(1, 1, 3) = 4, DBL = 2
J = -3746E 02 ; LGL(12, -2) = .FALSE.
CPX = (7.32D-2, 3) ; K = 0, ARRAY = 0,
0, 14, 3.71*
```

or

```
R(55, -2) = 55.349384531062851907
J = 9
BOOLE = T; FLAG = FORTRAN4
C = (2, 7.08364724286E-03)
ARRAY = 1.0
        2.718281828459046
        3.141592653589793
        -3944483

*END*
```

The *END* at the end of the second example is the form of end record generated by the OUTPUT statement, and enables the user to separate OUTPUT records into "files" that can later be processed by one or more INPUT statements. For example

```
OUTPUT (4) A, B, C, *, ARRAY, *, CPX, J(3), *
```

generates records that can be processed by three separate INPUT statements.

FORMAT Statement

The FORMAT statement is used to specify the conversion to be performed on data being transmitted during formatted (BCD) input/output or DECODE/ENCODE operations. It is nonexecutable and may be placed anywhere in the program. In general, conversion performed during output is the reverse of that performed during input. FORMAT statements are expressed as

FORMAT ($S_1, S_2, S_3, \ldots, S_n$)

where

$n \geq 0$

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

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

where

$m \geq 0$.

r is a repeat count as described below.

S_j is as described above; in other words, repetitions may be nested (to ten levels).

The commas between the S_i (and S_j) are optional except where ambiguities would arise from not separating specifications. In the absence of a comma, the compiler attaches as much as possible to the left-hand specification. For example, the specifications

```
I23F27.13X
```

will be interpreted as

```
I23 , F27.13 , X
```

and not as

```
I2 , 3F27.1 , 3X
```

To obtain the latter interpretation, the commas are required.

Every FORMAT statement should be labeled so that references may be made to it by formatted input/output statements. An entire FORMAT (the parentheses and the items they enclose) may be stored in an array variable through the use of assignment statements or input statements. In this case, as described under "FORMATs Stored in Arrays", the array itself is referenced by the input/output statements.

Format specifications describe the kind or type of conversion to be performed, specific data to be generated, scaling of data values, and editing to be executed. Each integer, real, double precision, or logical datum appearing in an input/output list is processed by a single format specification, while complex data are operated on by two consecutive format specifications. Format specifications may be any of the following forms:

rFw.d	rIw	rZw	iX
rEw.d	rLw	rMw	Tw
rDw.d	rAw	r's'	iP
rGw.d	rRw	nHs	r/

where

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

r is an optional, unsigned integer[†] that indicates that 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
```

[†]See also "Adjustable Format Specifications".

w is an optional unsigned integer[†] that defines width in characters (including digits, decimal points, algebraic signs, and blanks) of the external representation of the data being processed. If w is not present in a specification, the size of the external field depends on the characteristics of the data and the type of conversion performed. This is discussed individually under each specification.

d for F, E, D, and input G specifications, is an optional, unsigned integer[†] that specifies the number of fractional digits appearing in the magnitude portion of the external field. If d is not present, its value is assumed to be zero, and the decimal point character preceding it should not appear. That is, Ew.0 and Ew are equivalent.

For output 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 external field; therefore, its value should not be zero.

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

s is a string of the characters acceptable to the FLAG processor (see Chapter 1).

i is a signed integer[†] (plus signs are optional). The function of i is described under X and P specifications.

F Format (Fixed Decimal Point)

Form:

rFw.d

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. Double precision values are converted with full precision if sufficient width is specified by w, and the value of d allows for the appropriate number of digits in the fractional portion of the field.

Output. Internal values are converted to real constants, rounded to d decimal places with an overall length of w. The field is right justified with as many leading blanks as necessary. Negative values are preceded with a minus sign. Consequently, for the specification F11.4,

273.4	is converted to	273.4000
7	is converted to	7.0000
-.003	is converted to	-.0030
-442.30416	is converted to	-442.3042

When no width is specified (i.e., w is not present), the converted field contains only the number of digits necessary to express the value, plus one blank to the right of the field. Therefore, for the specification F.1,

349.5203	is converted to	349.5b
70000	is converted to	70000.0b
-22	is converted to	-22.0b

and for the specification 2F.4, the output list

.03359, -67	is converted to	.0336b-67.0000b

where b represents the character blank.

If a value requires more positions than are allowed by the magnitude of w, only w digits will appear, and the digits lost will be from the left or most significant portion of the field. This is not treated as an error condition. Thus, for the specification F4.4,

-1.22315	is converted to	2232
432034.	is converted to	0000

In order to insure that such a loss of digits does not occur, the following relation must hold true:

$w \geq d+2+n$

where n is the number of digits to the left of the decimal point.

[†]See also "Adjustable Format Specifications".

Input. Input strings may take any of the integer, real, or double precision constant forms discussed under "Numeric Input Strings". Each string will be of length w with d characters in the fractional portion of the value. 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. For the specification F10.3,

33	is converted to	.033
802142	is converted to	802.142
.34562	is converted to	.34562
-7.001	is converted to	-7.001

If the width w is not specified, conversion starts with the first non-blank character in the input string and ends with the first comma or blank that follows a digit or a decimal point. The comma or blank is bypassed before conversion of the next field begins. For the specification 2F.2, the string

333,.003

is converted to the values

3.33 .003

E Format (Normalized, with E Exponent)

Form:

rEw.d

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. Double precision values are converted with full precision if sufficient width is specified by w and the value of d allows for the appropriate number of digits in the fractional portion of the field.

Output. Internal values are converted to real constants of the forms

.ddd...dE ee

.ddd...dE-ee

where ddd...d represents d digits, while ee or -ee is interpreted as a multiplier of the forms

$10^{\pm ee}$

Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right justified and preceded by the appropriate number of blanks. The following are examples for the specification E15.8:

90.4450	is converted to	.90445000E 02
-435739015.	is converted to	-.43573902E 09
.000375	is converted to	.37500000E-03
-1	is converted to	-.10000000E 01
.2	is converted to	.20000000E 00
0.0	is converted to	.00000000E 00

When the width w is not present in the format specification, the converted field contains only the number of characters necessary to express the value of the data, plus one blank to the right of the field. If the specification 2E.5 is used, the output list

-774.119, 1.00001977

is converted to

-.77412E␣03␣.10000E␣01␣

where ␣ represents the character blank.

The field, counted from the right, includes the exponent digits, the sign (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). If a width specification is of

insufficient magnitude to allow expression of an 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.

Examples:

Value	E11.4	E8.4	E6.4
-2013.55	-.2014E 04	2014E 04	14E 04
.361887	.3619E 00	3619E 00	19E 00
.000134	.1340E-03	1340E-03	40E-03

To prevent a loss of this kind, it is necessary to ensure that the relation

$w \geq d+6$

is satisfied by the specification. Note that the above feature can be used intentionally to obtain the exponent field, which is an indication of magnitude range for any datum. For example, for the specification E3.0,

60255.034 is converted to 05

0.0000072 is converted to -05

Input. The discussion "Numeric Input Strings" contains a description of the forms permissible for strings of input characters. Conversion is identical to F format conversion. In particular, input fields for conversion in E format need not have exponents specified.

Examples:

Input Value	Specification	Converted to
-113409E2	E9.6	-11.340900
-409385E-03	E.2	-4.09385
849935E-02	E10.5	.0849935
6851	E.0	6851.0

First, the decimal point is positioned according to the specification; then, the value of the exponent is applied to determine the actual position of the decimal point. In the first example, -113409E2 with a specification of E11.6 is interpreted as -.113409E02; which, when evaluated (i.e., $-.113409 \times 10^{2}$), becomes -11.340900.

D Format (Normalized, with D Exponent)

Form:

rDw.d

This format is similar to E format, with the exception that for output, the character D will be present instead of the character E. For example,

for E12.6, -667.334 is converted to -.667334E 03

and

for D12.6, -667.334 is converted to -.667334D 03

Input under D format is the same as for E and F formats.

G Format (General)

Form:

rGw.d

G format 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, as with all other formats, 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. G format is very useful in a widthless form. When so used, the equivalent formats shown above become widthless also (see "Numeric Input Strings").

Output of Real and Double Precision Data Under G Format. The form of output conversion used with real and double precision values depends on the magnitude of the values. G format attempts to express numbers in the most natural way; that is, they are expressed in F format whenever possible, but in E format for values that are too large or too small. Specifically, d is interpreted as indicating the number of significant digits desired, and this is exactly the number of digits that will be output. If the value of the number is such that it can be expressed by placing the decimal point anywhere within or at either end of those d digits, that is what will be done, and no exponent will be appended. If, however, preceding or trailing zeros would be required to express the value correctly, F format will not be used; instead the number will be normalized and output with a following exponent.

To express this algebraically, let M represent the magnitude of the value to be output (rounded to d significant digits). Select an integer i such that

$$10^{i-1} \leq M < 10^{i} \qquad (\text{if } M = 0.0, \text{ then } i = 0)$$

Assuming a specification of Gw.d, let n = w-4 and m = d-i. Then, if $0 \leq i \leq d$, conversion takes place according to the specification

Fn.m,4X

If i is less than 0 or greater than d, the specification used is

Ew.d

Note that when F format is used, 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 line up underneath each other in a more readable way. The following examples illustrate the effect of G format output on values of various sizes:

Value	G 10.3	G 10.1
.02639	.264E-01	.3E-01
.2639	.264	.3
2.639	2.64	3.
26.39	26.4	.3E 02
263.9	264.	.3E 03
2639.	.264E 04	.3E 04

Note that the choice of F or E format is independent of the value of the width w. If w is not large enough, digits are lost at the left as in other numeric conversions. To ensure that this will not happen, the following relation should hold true:

$w \geq d+6$

When no width is specified, the number will be followed by a single blank; values output in F form will not be followed by four blanks.

Scale factors (see "P Specifications") apply to G format only when the E form is used, not when the F form is used. This has the effect that all values output in G format are unchanged (except for rounding). It also has the effect that values output in F form with a P scale factor cannot subsequently be input using the same FORMAT; the scale

factor will take effect during input but not during output. Thus, the new value will be different from the old by a power of 10.

Note that the rounding applied to M (above) to determine whether to use E or F format is not necessarily the same rounding that is applied when the number is actually output. Consider the following case:

```
    PRINT 5, 99.76
5   FORMAT(1P, G.2)
```

In principle, F form is to be used if the value lies in the range $.1 \leq M < 100$. The value 99.76 does lie in this range, but when rounded to two digits it becomes 100., which is outside the range; so E form is used. First the unrounded value of M is normalized (.9976E 02), then the P scale factor is applied (9.976E 01), and finally this value is rounded giving 9.98E 01, which is the way it is printed. If the first rounding had been used throughout, the final value would have been 1.00E 02, which is less accurate.

I Format (Integer)

Form:

rIw

Integer, real, double precision, or either part of complex data may be processed by this form of conversion. If the width specification w is of sufficient magnitude, real and double precision values are converted in full precision. In other words, values greater than the maximum permissible size of integer data may be processed, without the truncation of the most significant digits that is normally associated with integer operations.

Output. Internal values are converted to integer constants. Real and double precision data are truncated to integer values; however, the integers may contain as many digits as are specified by w. Negative values are preceded by a minus sign, and the field will be right-justified and preceded by the appropriate number of blanks. The specification I6 implies that

273.4	is converted to	273
7	is converted to	7
-.003	is converted to	0
-44204.965	is converted to	-44204

The converted field occupies the minimum number of positions required to express the data value whenever w is unspecified. This minimum number of digits is followed by one blank. For example, for the specification 5I the output list

345.9, 70000, -2, -.999, 3030.3030

is converted to

345␣70000␣-2␣0␣3030␣

where ␣ represents the character blank.

If the magnitude of data requires more positions than is permitted by the value of the width w, only w digits appear in the external string, and the digits lost are the most significant. This is not treated as an error condition. Thus, for the specification I2,

-778801 is converted to 01

Input. External input strings may take any of the forms discussed under "Numeric Input Strings" and conversion will be identical to F format processing, with the exception that fractional portions of a value are lost through truncation. As noted above, however, the most significant digits will not be truncated. For example, the input field

45700000000000000000000000.942

processed by an I (widthless) format, into a real or double precision variable, would produce the internal value

4.57×10^{24}

L Format (Logical)

Form:

rLw

Only logical data may be processed with this form of conversion; any other data type causes an error to occur.

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. For examples, using the specification L4,

.TRUE. is converted to ␣␣␣T

.FALSE. is converted to ␣␣␣F

where ␣ represents the character blank.

Specifications in which w is undefined will cause the following conversions:

.TRUE. is converted to T␣

.FALSE. is converted to F␣

Input. If a width is specified, 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, except for commas, which terminate the input string (see "Comma Field-Termination"). For example, the following input fields, processed by an L7 format, have the indicated values:

True	False
T	F
TRUE	FALSE
.TRUE.	.FALSE.
RIGHT	READ
STAFF	LEFT
24T+T42	(blank)

For widthless logical input, the field terminates at the first comma or non-leading blank. In other words, if the first non-blank character is a comma, it terminates the field; if it is not a comma, the next blank or comma will terminate the field. The first T or F encountered within the field determines the value. If neither a T nor an F appears, the field has the value "false". As above, characters appearing between the first T or F and the blank or comma are ignored.

A Format (Alphanumeric)

Form:

rAw

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 "Storage Allocation Statements", Chapter 7). Assuming standard size specifications, the following examples illustrate the form of A format conversion:

Data Type	Internal Binary/Hexadecimal	Aw	External String
integer, real, or logical	1100 1001 1101 0101 1110 0011 0101 1100 C 9 D 5 E 3 5 C	A4	INT*
		A2	IN
		A6	␣␣INT*
		A	INT*
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	␣␣␣DOUBLE=2
		A	DOUBLE=2

where ␣ represents the character blank.

As with all other format conversions, complex data are treated either as two real or as two double precision values. In each of the examples above, the first A format specifies exactly the number of characters required to express the data fully, and therefore has the same effect as the widthless form. Normally, alphanumeric information is used with integer variables. In the examples, note that when the magnitude of w does not provide for enough positions to express the data value completely, the external field is shortened from the right (least significant) portion. This is not treated as an error condition. When w has a value greater than necessary, the external character string is preceded by the appropriate number of blank characters.

When the field width is not specified, the external character string consists of only the number of positions necessary to fully express the character value of the data. The external character string is not followed by a blank.

Alphanumeric conversions are normally used to output Hollerith information that has been created in one of the following ways:

1. Previously input using an alphanumeric format (A or R)

2. Using a literal constant (i.e., in a DATA statement, or passed as an argument)

It is not recommended that this form of conversion be used with random numeric values created other than as above. The reason for this is that not all the 256 possible characters that can be produced can actually be printed. The non-printable characters may, however, be useful in other contexts (e.g., on cards, or in ENCODE operations).

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 list item is filled with the rightmost characters. For example, if the list item is integer, and the specification A10 is used,

ABCDEFGHIJ is converted to GHIJ

However, 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. Consequently, when the list item is double precision and the field specification is A6,

UVWXYZ is converted to UVWXYZbb

where b represents the character blank.

Naturally, if the width has a value equal to the number of characters associated with the data type of the list item, the list item is completely filled with the external field.

Widthless specifications cause the list item to be filled by the next n characters from the input string, where n is the number of characters associated with the data type of the list item. If a list contained references to a real variable, an integer variable, and a double precision variable, in that order, and a field specification of 3A were used, processing would be in the following manner:

ABCDEFGHIJKLMNOP

is converted to

ABCD EFGH IJKLMNOP

A general rule for this type of conversion is that internal values are considered to be left-justified, while external fields are considered to be right-justified.

R Format (Alphanumeric, Right-Justified)

Form:

rRw

This form of conversion is similar to A conversion, but the rule of internal justification is reversed. In other words, internal values are considered to be right-justified with leading binary zeros, whereas with A format they are left-justified with trailing Hollerith blanks.

Output. When the size of w is insufficient to allow expression of the complete internal value, R format takes characters from the rightmost (least significant) portion of the internal value. In all other respects it is identical to A format output. This difference is illustrated in the examples at the top of the following page.

Data Type	Internal Character Value	w	A Format	R Format
integer, real, or logical	INT*	4	INT*	INT*
		2	IN	T*
		6	␣␣INT*	␣␣INT*
		none	INT*	INT*
double precision	DOUBLE=2	8	DOUBLE=2	DOUBLE=2
		6	DOUBLE	UBLE=2
		10	␣␣DOUBLE=2	␣␣DOUBLE=2
		none	DOUBLE=2	DOUBLE=2

where ␣ represents the blank character.

Input. As on output, R format differs from A format only when the specified width (w) is less than the number of characters associated with the type of the input list item. In this case, R format fills the least significant (rightmost) portion of the list item with w characters from the input string, preceded by enough binary zeros to fill the remaining portion. In other words, R format right justifies the characters and inserts leading binary zeros, while A format left justifies the characters and inserts trailing Hollerith blanks. For example,

List Item Data Type	External String	w	Internal after A Conversion	Internal after R Conversion
integer, real, or logical	XYINT*	4	XYIN	XYIN
		6	INT*	INT*
		2	XY␣␣	zzXY
		none	XYIN	XYIN
double precision	85DOUBLE=2	8	85DOUBLE	85DOUBLE
		6	85DOUB␣␣	zz85DOUB
		10	DOUBLE=2	DOUBLE=2
		none	85DOUBLE	85DOUBLE

where

␣ represents the Hollerith character blank and

z represents eight binary zeros.

Note that the Hollerith character zero is not represented internally as eight binary zeros. Consequently, if the external field

00ABAB

were processed by the format specifications A4,R2 into two integer variables, the resulting values would be the Hollerith constants 4H00AB and 2RAB, which are not equivalent. For input as true right-justified integers, R format should be used.

Z Format (Hexadecimal)

Form:

rZw

Z conversion is similar to R conversion, except that the internal data is processed 4 bits at a time instead of 8, and the external field consists of hexadecimal digits, which are:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Output. Internal binary values are converted to hexadecimal digit strings at the rate of 4 bits per digit. The number of characters produced by an item depends on the number of words of storage allocated for that type of item (see "Storage Allocation Statements", Chapter 7). For example, an integer produces 8 digits, a double precision number, 16.

If w is not specified large enough, the leftmost digits are lost, as in other numeric formats. If w is larger than the number of positions necessary to express the data, the digits are right-justified in the field, with preceding blanks.

When field width permits, all of the digits in an item are output, including leading zeros.

When w is not specified, the full number of digits necessary to express the value is output, followed by a blank. The blank is to facilitate subsequent rereading of the value (see below).

Examples:

Data Type	Internal Binary/Hexadecimal								Zw	External String
integer, real, or logical	0000 0	0000 0	0000 0	1000 8	1110 E	0011 3	0101 5	1100 C	Z8	0008E35C
									Z6	08E35C
									Z10	␣␣0008E35C
									Z	0008E35C
double precision	0100 4	0001 1	0011 3	0010 2	0100 4	0011 3	1111 F	0110 6	Z16	413243F6A8885A30
									Z11	3F6A8885A30
	1010 A	1000 8	1000 8	1000 8	0101 5	1010 A	0011 3	0000 0	Z18	␣␣413243F6A8885A30
									Z	413243F6A8885A30

where ␣ represents a blank character.

Input. When the width w is larger than necessary (i.e., when its magnitude is greater than the number of digits associated with the data type of the corresponding list item), the list item is filled with the rightmost characters in the field.

When w is too small, the digits are right-justified in the list item, as with R format. As usual, when the width exactly corresponds to the number of digits associated with the list item, the item is completely filled with the external field.

There is, however, a significant difference between Z and R format on input. Z format is a numeric format, not alphanumeric. Therefore, commas may be used to terminate a hexadecimal input string. Furthermore, the length of a widthless Z input string is not dependent on the size associated with the list item; a widthless hexadecimal input string terminates at the first comma or non-leading blank, like all other numeric formats. Excess digits will be lost at the left. (Note that, when w is specified, blanks are treated as zeros.)

The following are examples of Z format input (assuming an integer list item):

External Input Field	Format	Internal Hexadecimal Value
3A70049B	Z5	0003A700
␣␣␣3A7␣␣	Z8	0003A700
␣D68,47019	Z8	00000D68
DCBA987654321	Z12	98765432
52CA91,	Z	0052CA91
123456789ABC,	Z	56789ABC
␣␣49␣␣3	Z	00000049

where ␣ represents a blank.

M Format (Machine Dependent)

M format is intended primarily for output. It provides a machine-independent method of dumping information in the format most appropriate to the machine on which the program is running. Thus, on an octal machine it would

be interpreted as O format, and on a character machine, as A format. On the Sigma 5/7, it is interpreted exactly the same as Z (hexadecimal) format. Thus, it could also be used for input, though this is not recommended.

H Format (Hollerith)

Form:

nHs

where

$n \leq 255$

Output. The n characters in the string s are transmitted to the external record. For instance,

Specification	External String
1HE	E
8HbbVALUE:	bbVALUE:
5H$3.95	$3.95
9HX(2,5)b=b	X(2,5)b=b

where b represents the character blank.

Care should be taken that the character string s contains exactly n characters, so that the desired external field will be created, and so that characters from other format specifications are not used as part of the string.

Input. The n characters in the string s are replaced by the next n characters from the input record. This replacement occurs as shown in the following examples:

Specification	Input String	Resultant Specification
3H123	ABC	3HABC
10HNOWbISbTHE	bTIMEbFORb	10HbTIMEbFORb
5HTRUEb	FALSE	5HFALSE
6Hbbbbbb	RANDOM	6HRANDOM

where b represents the character blank. This feature can be used to change the titles, dates, column headings, etc., that are to appear on an output record generated by the H specification.

If n is not present, its value is assumed to be 1.

' Format (Hollerith)

This is an alternate format for Hollerith transmission similar to that done by H format. This has the advantage of not requiring the characters in the string to be counted.

Form:

's'

The string s may contain not more than 255 characters. Any Hollerith characters may appear (see Chapter 1); however, note the restrictions below concerning the ' character. A repeat count, r, may optionally precede the specification.

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

Within a ' string the ' character is represented by two adjacent ' characters; thus, 'I' 'LL TAKE FIVE is output as

I'LL TAKE FIVE

Input. The characters appearing between the quotes are replaced by the same number of characters taken sequentially from the input string. Therefore, if the specification

'VECTOR'

is used to process the input field

MATRIX

the specification itself is changed to

'MATRIX'

Blanks in FORMAT statements are significant only in H and ' specifications.

X Specification (Skip; Space or Backspace)

The form of the X specification is

iX

This specification causes no conversions to occur. Instead, it causes i positions of the external field to be "skipped". If i is positive, it has an effect similar to that of a space bar on a typewriter; if it is negative, it has an effect similar to that of the backspace control on a typewriter. In particular, an attempt to backspace beyond the beginning of a record is equivalent to backspacing to the beginning of the record.

Output. For positive values of i, the next i positions in the output record will be blanks (normally, however, see below). In other words, a field of i blanks will be created. For example, the specifications

'WXYZ' , 4X , 'IJKL'

generate the following external string:

WXYZbbbbIJKL

where b represents the character blank.

A negative value of i causes processing to "back up" in the record. The next field will then begin |i| characters to the left, assuming that this is not beyond the beginning of the record. For example, the specifications

'FORTRAN' , -3X , 'KNOX'

are equivalent to the specification

'FORTKNOX'

Note that when either backing up or moving forward by means of an X specification, characters that may have been previously produced in the positions being skipped are not destroyed. Thus, in the example given above under X output, it is not necessarily true that the specification 4X will produce four blanks. It will, however, if no other characters have been generated in those positions, since all output records are initially set to blanks.

The ability to specify a negative count in an X specification makes it possible to backspace over the blank that is produced at the end of external fields by widthless numeric formats (i.e., D, E, F, G, and I). For example, for K = 13 and Q(13) + 350.8, the statements

```
      PRINT 5,  K  ,  Q(K)

    5 FORMAT( 'Q('  ,  I  ,  -X  ,  ') = '  ,  F.2)
```

generate the string

Q (13) = 350.80

As illustrated in the above example, if i is not specified it is assumed to be 1. Thus, the following specifications are equivalent:

XXXX

4X

Input. The next i characters from the input string are ignored whenever i is positive (that is, they are skipped). For example, with the specifications

F5.3, 6X, I3

and the input string

76.41IGNORE697

the characters

IGNORE

will not be processed.

Negative values of i cause |i| characters from the input string to be processed again. Consequently, the specifications

I3, -1X, E4.1

and the string

123456

are equivalent to

I3, E4.1

and the string

1233456

T Specification (Tab)

The form of the T specification is

Tw

This specification causes processing (either input or output) to begin at character position w in the record, regardless of the position in the record that was being processed before the T specification. It functions exactly like an X specification; no transfer of data occurs. For example, the following FORMATs are equivalent:

```
1 FORMAT( 5X , A8 , -2X , I7 )
2 FORMAT( T6 , A8 , T12 , I7 )
```

It can be seen from the above example that it is permissible to tab either forward or backward. Furthermore, a T specification provides a capability that an X specification does not, namely that of tabbing to a given print position when widthless formats are being used and the character position is thus unknown. For example, to print (or read) three columns of numbers beginning in positions 1, 21, and 41, the following FORMAT statement could be used:

```
3 FORMAT(G.7, T21, G.7, T41, G.7)
```

Note that backward tabbing can cause previously output information to be overwritten, or previously read input to be processed again.

As with X specifications, it is not possible to tab to a position previous to the beginning of the record.

If no w is given, it is assumed to be 1. That is, T is the same as T1.

P Specification (Scale Factor or Power of 10)

The form of the P specification is

iP

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 forms

10^{i} for output

and

10^{-i} for input

At the beginning of each formatted input/output operation, before any processing occurs, the scale factor is set to zero. 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. If a FORMAT is re-scanned within a single input/output operation due to the number of items in a list (see "FORMAT and List Interfacing"), the value of the scale factor is not reset to zero.

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

Output. The value of the list item is scaled by the multiplier 10^i. This scaling causes the decimal point to be shifted right i places. On D- and E-type conversions, the exponent field (±ee) is correspondingly reduced by 1. Thus, for D- and E-type output, the external number is equal to the internal value (except for rounding), while for F format output it is not (unless i is 0). Scale factors do not affect numbers whose value is zero. The following examples illustrate output scaling:

Format	External field when internal value is:			
	2.71828	-2.71828	0.00000	0.09999
2PF10.3	271.828	-271.828	.000	9.999
1PF10.3	27.183	-27.183	.000	1.000
0PF10.3	2.718	-2.718	.000	.100
-1PF10.3	.272	-.272	.000	.010
-2PF10.3	.027	-.027	.000	.001
-3PF10.3	.003	-.003	.000	.000
-4PF10.3	.000	-.000	.000	.000
2PE14.3	27.183E-01	-27.183E-01	.000E 00	99.990E-03
1PE14.3	2.718E 00	-2.718E 00	.000E 00	9.999E-02
0PE14.3	.272E 01	-.272E 01	.000E 00	.100E 00
-1PE14.3	.027E 02	-.027E 02	.000E 00	.010E 01
-2PE14.3	.003E 03	-.003E 03	.000E 00	.001E 02
-3PE14.3	.000E 04	-.000E 04	.000E 00	.000E 03
-4PE14.3	.000E 05	-.000E 05	.000E 00	.000E 03

The examples for E conversion above are similar to those that would result from D conversion and E-type G conversion. When G conversion uses the F form, however, scale factors do not apply. Thus, a number output in G format always represents the internal value.

Note that when a scale factor is in effect, output rounding takes place after the scaling has been performed. In the case of E format, this may cause additional scaling to be required, as shown above in the output of 0.09999. Note the discontinuity in the way the exponent changes.

Input. During F, E, D, and G input conversions, if the input string contains an exponent field, the scale factor has no effect. However, when the input string does not contain an exponent field, the value of the external field is scaled by 10^{-i}; that is, the decimal point is moved left i places. 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
-71.436E 00	3P	-71.436
	-1P	-71.436

It can be seen that, on both input and output, if the external number has an exponent specified, it is equal to the internal value; if it does not, then

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

Once a scale factor has been established during an input/output operation, it remains in effect throughout the operation, unless redefined by an additional P specification. To reset the scale factor to zero, it is necessary to write a 0P specification. For the list

A, K, X, B

and the FORMAT

FORMAT(2(F.3,2P),E12.4,-2P)

A, K, and B are all converted using the F.3 format specification, but all three have different scale factors in effect, as illustrated below:

List Item	Effective Format Specification
A	F.3
K	2PF.3
X	2PE12.4
B	-2PF.3

When i is not specified, its value is assumed to be zero. Therefore,

P is equivalent to 0P

/ Specification (Record Separator)

The form of the / specification is

r/ or /

Each slash(/) specified causes another record to be processed. In the case of contiguous slash specifications (i.e., ////.../ or r/), 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. Note that 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, a termination occurs, and the final or sixth record, which is also blank, is output.

Note that the processing of Z in steps 5 through 8 is equivalent to processing with the statement

```
10 FORMAT (F5.3,//)
```

since the specification I13 was not utilized.

The original FORMAT statement could also have been written as

```
10 FORMAT (F5.3, 2/I13)
```

or

```
10 FORMAT (F5.3, 2/, I13)
```

both of which would cause identical effects.

The two statements

```
  WRITE (M, 4) X
4 FORMAT (3/E.4/)
```

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

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

```
  READ (M, 4) X
4 FORMAT (3/E.4/)
```

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

Parenthesized FORMAT Specifications

Within a FORMAT statement any number of specifications may be repeated by enclosing them in parentheses, preceded by an optional repeat count, in the form shown on the following page.

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

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

```
3 FORMAT (3(A4, F. 2, 3X), 3I)
```

is equivalent to

```
3 FORMAT (A4, F. 2, 3X, A4, F. 2, 3X, A4, F. 2, 3X, 3I)
```

There is no limit to the number of repetitions of this form that can be present in a FORMAT statement.

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

The following are additional examples of repetitive specifications:

```
  34 FORMAT (4X,2(A8,X,7G.3),I4,3(D,L5))
1125 FORMAT (/,R4,F.7,5(E14.8,2/),E14.8)
   8 FORMAT (2(I8,2(3X,F12.9),F12.9),A16)
```

In the last example above, repetitions have been nested. Nesting of this type is permissible to a depth of ten levels.

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 the first time 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".

Adjustable FORMAT Specifications

The adjustable FORMAT specifications feature often eliminates the need to write a great number of FORMAT statements in order to handle slightly different situations. Furthermore, it facilitates the input of records whose form is highly variable, and which could not be processed without this feature.

Any of the quantities r, w, d, or i (see "FORMAT Statement") may be replaced by the letter N in a format specification. When an N is encountered, its value is obtained from the next input or output list item. The letter N is merely a form of specification and does not conflict with any variable, subprogram, etc., whose identifier may be N. Also, there is no limit to the number of N characters that may be used in a FORMAT statement or to the number of quantities replaced by N in a format specification. For example,

```
32 FORMAT (NX,FN.4,N(3X,E.5),NP,NGN.N)
```

is a valid statement, and seven values will be taken form the list.

The following set of rules defines the manner in which the value of N must be specified in a list and the way in which the values are utilized:

1. Integer, real, double precision, or either part of complex data may be supplied as values for N. Non-integer data will be truncated to integer value.
2. w and d (width and decimal point) specifications may be replaced only by N, whereas r (repeat count) and i (skip or scale factor count) specifications may be replaced by N or -N.
3. The resultant value (negated if preceded by a minus sign) may be negative only when N is used to replace i.
4. When N appears one or more times in a single specification, its values must appear sequentially in the list and prior to the items (if any) that are to be processed by the specification. An example is the list

   ```
   3,4,1,A,B,C,12,-2,D
   ```

   and the statement

   ```
   3 FORMAT (NEN.N,NX,NP,G14.8)
   ```

   which are equivalent to the list

   ```
   A,B,C,D
   ```

   and the statement

   ```
   3 FORMAT (3E4.1,12X,-2P,G14.8)
   ```

5. Whenever N is used with a specification that is enclosed in repetition-type parentheses (see "Parenthezised Format Specifications"), one value must be supplied for each repetition of the specifications enclosed. Consequently, the difference between the following two examples should be noted:

7, A, B, C and 3FN. 2	7, A, 7, B, 7, C and 3(FN. 2)
are equivalent to	are equivalent to
A, B, C and 3F7. 2	A, B, C and 3F7. 2

6. In the above example, it was noted that in the specification 3FN.2, one value of N is required, regardless of the value of the repeat count; whereas, in 3(FN.2), the number of values required for N is equal to the repeat count. The same rule can be extended to include repeat counts whose values are zero:

   a. When the repeat count (r) of a single specification is replaced by N and its value is zero, any Ns appearing in that specification must be supplied. For example, the following combination of list and FORMAT,

      0,4,Y and NG20.N,F8.4

      are equivalent to

      Y and F8.4

   b. However, when the repeat count of a parenthesized group is replaced by N and its value is zero, all the specifications appearing within the parentheses are bypassed, including any Ns that may appear. Thus,

      0,Y and N(G20.N),F8.4

      are equivalent to

      Y and F8.4

   In both of the above examples, no value was supplied for the G specification; however, enclosing the specification within parentheses can be used to determine whether or not the value of N will be supplied.

   The ability to specify zero repeat counts in this way gives the programmer the facility of selecting or skipping certain specifications within a FORMAT statement. For example,

```
T = 0
IF (BOOLE) T = 1
F = 1 - T
PRINT 17, BOOLE, T, F
17 FORMAT(L1, N(3HRUE), N'ALSE')
```

   outputs the strings TRUE or FALSE depending on the value of BOOLE. Note that although an N cannot replace the n in an H specification, the form shown in statement 17 above can be used.

7. The value of N may be supplied by an expression in either an input or an output list, but an expression used for this purpose in an input list is not considered to be a true input list item.

As an example of the flexibility provided by adjustable format specifications, consider the statements

```
READ(101,205) K, K, (A(J), J=1,K), CODE
205 FORMAT( I , NE , A4 )
```

The value input for K defines not only the number of values to be input into the array A, but also the number of conversions to be performed by the E specification. At the same time, the alphanumeric value of CODE can be contiguous to the last field input into A, regardless of the number of such fields. Thus, all the following input records can be correctly processed by the above statements:

```
1, 67.49,HOPA
5 -14.3 37 .09711623 0 3E12 JASU
,NONE
```

This example illustrates not only adjustable format specifications, but also widthless formats and comma field termination (see below).

Numeric Input Strings

The permissible kinds of input strings that may be processed by numeric conversions are exactly the same for F, E, D, G, and I conversion. Any field that can be read using one of these formats can be read using any of the others. In other words, numbers for input with E format need not have exponents, numbers for input with I format need not be integers, etc.

A numeric input string consists of a string of digits with or without a leading sign, a decimal point, and/or a trailing exponent. An exponent is normally specified as

E±e

where the plus sign is optional and e is a one- or two-digit number. The form ±e is also accepted (without the E), in which case the plus sign is not optional. Thus, a variety of forms may be used to express data for numeric input:

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

where the plus signs are optional except in an exponent field without an E (as described above).

When input fields contain no decimal point (as in the first column above) the decimal point is positioned according to the d in the format specification (as in Ew.d). If none is specified it is assumed to be zero. The decimal point is placed d positions to the left of the beginning of the exponent, or if no exponent is present, d positions to the left of the end of the field. Note that the exponent may begin with either a D, E, +, or -.

A D may be substituted for the E in an exponent field, with no change in meaning or value. It is not necessary to indicate that data is double precision, nor is it necessary to use a D format. Regardless of the format used or the form of exponent (if any), a numeric string will be converted with full double precision if the input list item to which it is to be assigned is double precision.

Any numeric type of list item may be used with any numeric type of format specification. If the list item is integer, the input value will be processed in floating-point, if necessary, and then converted to integer. When the I format specification is used (with any type of list item), the fractional portion of the value is lost.

A comma may be used to terminate any numeric field, as described below. Leading blanks are always ignored. The interpretation of embedded and trailing blanks depends on whether or not the format specification used is widthless (no width specified).

Widthless Numeric Input

The principle behind widthless input is that the field ends when the number is finished. A comma always indicates that the number is finished. A blank also indicates that the number is finished, if it is meaningful to finish the number at that point. Thus:

1. Leading blanks do not cause termination; they are ignored.
2. Any number of blanks may appear in the following places:
   a. Between the leading plus or minus sign and the first digit.
   b. Between the E and the plus or minus sign or first digit of the exponent.
   c. Between the plus or minus sign in the exponent and the first digit of the exponent.
3. A blank that follows a digit or decimal point terminates the field.
4. When a widthless (or any other) field runs off the end of the input record, the extra characters will be interpreted as blanks. Normally, a widthless format does not terminate until at least one non-blank character has been found. Special provision is made, however, to terminate widthless fields at the end of the record. Thus, any number of numeric values may be read from a blank record, and they will all be zero.

For clarity, numbers should generally be written without any embedded blanks. The first blank will then terminate the field. Although the terminating blank or comma does not affect the value of the number, it is considered part of the field it has terminated. Therefore, the next field begins with the character following the blank or comma.

The following is a typical widthless numeric input line consisting of eight values:

```
73  2E-4  .0007  -35.4  0  0  -16  27.08614E 12
```

The following is not a typical widthless numeric input line:

```
-  3E  +  2  +  3.7  -  4  17E  2  5-  03
```

but would be interpreted as five values, namely,

```
-300.          3.7        -4.       1700.      .005
```

Numeric Input with Width Specified

When a width is specified, the field terminates only when the width is exhausted or a comma is found. The following rules apply to blanks in numeric fields with a width specified:

1. Leading blanks are ignored, except that they are counted as part of the field width.
2. Once any nonblank character has been found, all blanks beyond that point are treated as zeros.
3. Any string of digits that is omitted has an assumed value of zero.

For a format specification such as F10.0, with no P scale factor, all the input strings in each of the columns below produce the value shown in the top line of the column. The first three lines in each column are typical numeric fields; the others are permissible, but less readable.

-.004	7.5E12	0
-4E-3	.75D+13	0.0
-.004	750E10	
- 4 - 4	75 E1	0 + 0
- . 4D	75 + 01	0E
-4 -8	. 75E16	+ -

Note, in the fourth example of the middle column above, that the exponent is interpreted as 10 rather than as 1, because the trailing blank is equivalent to a zero. Care should always be taken to assure that exponents are right-justified in their fields. Failure to do this is a common pitfall that can also be avoided by using comma termination and/or widthless formats.

Comma Field-Termination

Input strings being processed under control of F, E, D, G, I, or L specifications may be terminated at any point by the presence of a comma in the string.[†] In other words, whenever a comma appears in such an input string, the field currently being processed is considered ended, and no additional characters are converted. This termination occurs regardless of the value of w in the field specification. The comma is not processed, and the next field begins with the character following the comma. For example, the specification 2F13.3 and the string

```
3450,88412,
```

are equivalent to F4.3, F5.3 and

```
345088412
```

The string containing the commas would also be correctly processed by the specifications 2F.3 or 2F8.3.

Two contiguous comma characters indicate an empty field, which has the value zero. Therefore, for the specification 5I6, the string

```
303,,-1,,000450
```

is converted to the values

```
303  0  -1  0  450
```

[†]For consistency with symbolic input (via the INPUT statement), the characters semicolon, asterisk, and right parenthesis are also accepted as field terminators. Use of the comma is recommended, however.

and the string

```
0,,,,,
```

is converted to the values

```
0  0  0  0  0
```

The comma must, of course, fall within the field it is meant to terminate. For example, if the format specification F4.0 were used to process the input string

```
1234,
```

the value would already be terminated because of field width, and the comma would terminate the following field, giving it a value of zero.

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 be begun; 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, A, R, Z, M, or N 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, P, or /), it is processed whether or not another list item exists. Thus, for example, the statements

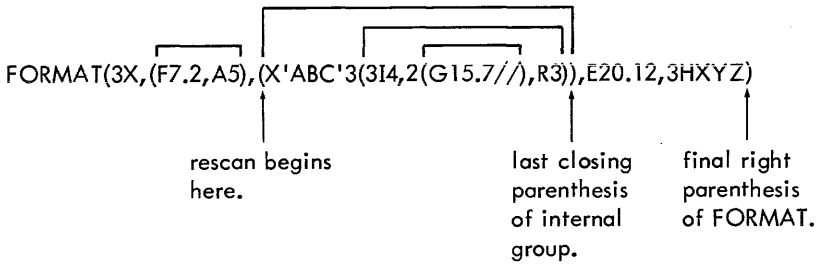
```
      WRITE(6, 12)
   12 FORMAT(///4HABCD)
```

   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 (including those to be used as values of N in adjustable specifications) 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, however, 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.



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. In particular, if the repeat count was specified with an N, a new value of N is not supplied when the rescan takes place; the old value is used. Thus for example, the statements

```
      PRINT 5, CODE, 5, (A(J), J=1,50)
    5 FORMAT( A4 / N(G20.8) )
```

are equivalent to the statements

```
      PRINT 5, CODE, (A(J), J=1,50)
    5 FORMAT( A4 / 5(G20.8) )
```

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, which 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 adjustable or conversion (see step 4 above) specification. If this condition is not met, the FORMAT statement will be processed, but an error will occur.

8. The same rules apply to DECODE and ENCODE operations as to READ and WRITE. The interpretations of multiple records in these cases is described under "Memory-to-Memory Data Conversion".

FORMATs Stored in Arrays

As mentioned previously, 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 or an assignment 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,

while

```
WRITE (4,R(1)) E,F,G
```

refers to a FORMAT statement whose location has been ASSIGNed to R(1).

If the variable M is an integer array, the following are two methods that may be used to store a FORMAT in M:

the external string

```
      (F8.5,4HNAIL,I3)
```

and the statements

```
      READ (N,90)(M(I),I = 1,4)
   90 FORMAT (4A4)
```

are equivalent to

the statements

```
      M(1) = 4H(F8.
      M(2) = 4H5,4H
      M(3) = 4HNAIL
      M(4) = 4H,I3)
```

Alternatively, M could be a dummy array corresponding to a literal constant argument (see "Arguments and Dummies").

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, while blank characters

are insignificant in all other specifications. 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.␣␣␣␣
M(2)	5,4H␣␣␣␣
M(3)	NAIL␣␣␣␣
M(4)	,I3)␣␣␣␣

which is not the desired result, since it is equivalent to the FORMAT

(F8.5, 4H␣␣␣␣, NA, I, L, I3)

where ␣ represents the character blank.

Even though a FORMAT may be quite short, such as

(I8)

it must be stored in an array. It may not be stored in a scalar variable, since a reference to a scalar variable (or an array element) will be treated as though the scalar were assigned the location of a FORMAT statement, rather than as if the scalar contained the FORMAT.

Extended Input/Output Capabilities

The statements described below under the headings

"Memory-to-Memory Data Conversion"

"Direct Input/Output"

"Random Access Input/Output Statements"

are optional features within the FLAG language. Programmers wishing to use any of these statements should ensure that the FLAG system available for their use includes these optional statements.

Memory-to-Memory Data Conversion

The statements ENCODE and DECODE are similar to formatted (BCD) WRITE and READ statements, respectively. In an ENCODE/DECODE operation, however, no actual input/output takes place; data conversion takes place between an input/output list and an internal buffer area. This buffer area is specified by the programmer and is usually an integer array. Whereas an external record has a certain physical length, the length of the simulated internal record in an ENCODE/DECODE operation may be specified by the programmer. When multiple records are specified by the FORMAT being used, records after the first record follow each other in memory in order of increasing storage address. These statements have the form

ENCODE(c, f, s, n) k	DECODE(c, f, s, n) k
or	or
ENCODE(c, f, s) k	DECODE(c, f,s) k

where

c defines the number of characters per internal record. It may be an integer constant or an integer non-subscripted variable.

f specifies a FORMAT statement. It may be the statement label of a FORMAT statement or the name of an array in which a FORMAT has been stored.

s indicates the first element or starting location of the internal buffer. It may be an array name, an array element, or a scalar variable.

n is an optional integer variable into which will be stored, upon completion of the operation, the number of characters actually processed (generated or scanned).

k is an input/output list of the usual form, and

a comma may optionally precede the list k.

Thus, the ENCODE/DECODE statements can be illustrated as

```
[ENCODE]
[DECODE]    (characters, format, start, count) list
```

Characters in the buffer are processed at the rate of four per word without regard to the type of the variable specified as the starting location. When a new record is begun, it starts at the first character following the previous record; in other words, at character c + 1. It is recommended that c be an integral multiple of four characters.

ENCODE Statement

The ENCODE statement causes the list items to be converted to BCD character strings, according to the FORMAT specified by f, and to be placed in storage beginning at location s.

If the number of characters generated by the FORMAT statement is greater than the specified size of the record, the extra characters are lost; they are not filled into the following record. If fewer characters are generated than are necessary to fill the record, it is filled out with trailing blanks. In fact, on ENCODE operations, as on WRITE BCD operations, the first thing done with each record is to fill it with blanks; this is done before any characters are stored (generated) into it.

For example, the following statements might be used to create, for later use, a FORMAT stored in the array M:

```
      K = 12
      L = 5
      ENCODE(12, 3, M) K, L
    3 FORMAT(2H(F , I , 1H. , I , 1H) )
```

The FORMAT so created would occupy the first three elements of M and would appear as

(F12␣. 5␣)␣␣␣

where ␣ represents the character blank.

DECODE Statement

The DECODE statement causes the character string beginning at location s to be decoded, according to the FORMAT specified by f, and stored into the items in list k.

As with formatted READ operations, if the FORMAT statement requires more characters from a record than are specified by the count (c), the extra characters are assumed to be blanks; they are not obtained from the next record. A new record is begun only when specifically requested by the FORMAT (see "FORMAT and List Interfacing").

If n is specified, it will be set to the number of characters scanned. When scanning with widthless formats, this can be very useful. The following example makes use of this feature:

```
      INTEGER KARD(80), DAVE/'DAVE'/
      READ 4, KARD
    4 FORMAT(20A4)
      DECODE(80, 5, KARD, NC) KODE, J
    5 FORMAT(A4, I)
      IF (KODE .EQ. DAVE) DECODE(80, 6, KARD) NC, J, (A(I), I=1, J)
    6 FORMAT(NX, NF)
```

The above statements could be used to read records of the form

```
DAVE 2, 1.75, 80.91
```

Note that, in the above example, the first DECODE statement is used to decide how to interpret the rest of the card. DECODE essentially provides the capability of "reading the card twice". ENCODE cannot be used in quite the same way because it initially fills its buffer with blanks.

Direct Input/Output

FLAG provides the facility to perform asynchronous input/output operations through the two library routines

BUFFER IN and BUFFER OUT

which have two major capabilities.

1. They permit the processing, both on input and output, of records of arbitrary length and format. Ordinary formatted READ and WRITE statements handle records only up to 132 characters in length, and unformatted, or binary, READ and WRITE statements process information that is intended for communication only with a FORTRAN environment. That is, in FORTRAN, binary information is considered to be a form of intermediate storage, an extension of computer memory, and as such, binary records have specialized control information and are broken into fixed-length physical records that comprise a logical record (see the XDS Sigma 5/7 FORTRAN IV Operations manual). The BUFFER IN and BUFFER OUT routines, however, process information exactly as specified, giving the programmer complete control over the data and enabling him to do such things as:

   interpret binary tapes produced on other machines or by other programs,

   read and write binary cards, and

   in conjunction with the ENCODE and DECODE statements, process long formatted records.

2. BUFFER IN and BUFFER OUT proceed in parallel with the program and other input/output operations. This enables the programmer to initiate an operation, continue with computation and other processing while the input/output is taking place, and to test the status of the BUFFER IN or BUFFER OUT operation at some later point in the program.

The BUFFER IN and BUFFER OUT subroutines are called in the following fashion:

CALL BUFFER IN(u, m, s, w, i, n)

and

CALL BUFFER OUT(u, m, s, w, i, n)

where

- u is an integer constant or an integer nonsubscripted variable that specifies the logical unit number of the device on which the operation is to be performed.
- m is an integer constant or an integer nonsubscripted variable that determines the mode of the operation; if m=0, the mode is BCD; otherwise, the mode is binary. (An integer 0 or 1 is customary.)
- s indicates the starting location of the internal buffer. Normally, s is the identifier of an array, but it may also be a scalar. It may be of any type, although integer is recommended for ease of manipulation.
- w specifies the number of words to be input or output, starting at s, and must be an integer constant or scalar of positive value.
- i is an integer scalar into which is dynamically stored an indication of the status of the operation. The status is indicated as follows:

  1 = operation incomplete

  2 = successful completion; no errors

  3 = end-of-file encountered

  4 = operation complete but error has occurred

- n is optional, but when specified is an integer scalar into which is stored, upon completion of the operation, the number of words actually input or output. It is not continuously set up "on the fly" as the operation is in progress. In general this count is significant only for BUFFER IN operations; in BUFFER OUT operations, n is normally equal to w (see below)

Thus, the BUFFER IN/BUFFER OUT calls can be illustrated as

CALL {BUFFER IN / BUFFER OUT} (unit, mode, start, words, indicator, count)

A BUFFER IN or BUFFER OUT operation always results in the processing of only one physical record. Data is read into, or written from, consecutive words in memory with no regard to the type of the variable specified as the starting location of the buffer. That is, the variable s merely specifies the starting location.

It is permissible to intermix asynchronous operations and standard READ/WRITE operations in any order and on any device, including intermixing on the same unit.

BUFFER IN

A call on BUFFER IN causes data to be read into memory from the specified unit, beginning at the location specified, in the specified mode. The actual number of words entered into memory is the minimum of w and n. That is, if more words are specified by w than are actually present in the physical record that is read, only the number of words appearing in the record are changed in memory, and that fact will be reflected by the value of n. However, if more words are present on the record than are specified by w, the extra words will be lost, and n will be greater than w.

When an end-of-file is read, i is set to 3; magnetic tape units will remain positioned immediately following the end-of-file. No data will be read into memory upon encountering an end-of-file.

The error status will be indicated when an irrecoverable error occurs. However, the data will be read into memory despite the error, and can be used if the programmer chooses to ignore the error.

Example:

The following statements could be used to list binary tapes, in hexadecimal, ten words per line, preceded by the record length:

```
      INTEGER BUFFER (5000)
    1 CALL BUFFER IN (5, 1, BUFFER, 5000, J, N)
    2 GO TO (2, 3, 4, 3, ), J
    3 M = MIN (N, 5000)
      PRINT 5, M, (BUFFER(K), K = 1, M)
      GO TO 1
    4 STOP
    5 FORMAT (/X, 'LENGTH = ' I / (X, 10Z10))
```

BUFFER OUT

A call on BUFFER OUT causes data to be written from memory, beginning with the specified location, in the specified mode. The number of words requested is always written, unless it is larger than the maximum size of a physical record on the device being used, in which case n will be less than w (it can never be greater). For example, an attempt to write 30 words on a card, in BCD, would result in n being set to 20.

If the indicator variable (i) has been set to indicate an error, an irrecoverable write error has occurred. The data has, nonetheless, been written on the specified device.

As mentioned above, records written in binary by BUFFER OUT are not the same as those produced by a binary WRITE statement. This has one important ramification: the BACKSPACE statement may not be used to backspace over binary records created by BUFFER OUT. It can be used to backspace over any kind of BCD records, but in binary records it expects to find the control words generated by a binary WRITE statement so that it may backspace over the entire logical record (which may consist of several physical records).

Note that the output produced in BCD by BUFFER OUT is virtually identical to that produced by a BCD WRITE statement using 'A' format. However, this does not include carriage control on printed output or any restrictions on the size of magnetic tape records.

Example:

The statements below could be used to read BCD cards and pack them onto binary cards.

```
      INTEGER M(40)
    1 READ (105, 2) M
    2 FORMAT (20A4)
      CALL BUFFER OUT (106, 1, M, 40, INDIC)
    3 IF (INDIC < 2) GO TO 3
      GO TO 1
```

Note that the reference to the array M in the READ statement causes two cards (40 words) to be read and causes the FORMAT to be scanned twice.

Random Access Input/Output Statements

FLAG allows full use of random access devices through utilization of the following two statements.

READ DISC u, s, k WRITE DISC u, s, k

where

- u is an integer constant or nonsubscripted integer variable whose value specifies the logical unit number of the disc.
- s is also an integer constant or nonsubscripted integer variable whose value defines the starting disc address (see below).
- k is an input/output list, as described previously.

Random access input/output operations are performed in binary, and therefore do not reference a FORMAT statement. They differ from the standard binary READ/WRITE statements, however, in two ways:

1. They refer to random access files rather than to sequential files. Consequently, the REWIND, BACKSPACE, and ENDFILE operations are not applicable to them.

2. Information is not thought of as being broken into unit records. Data is processed exactly as specified, with no control words or record boundaries. As many locations of the disc or drum are used as are required for the items specified in the input/output list. With a knowledge of the required sizes of various items the programmer is not bound by the binary READ/WRITE restriction that the data written by one WRITE statement must be input with one and only one READ statement.

   As an analogy, the disc may be thought of as a one-dimensional array, from which it is possible to select an element or group of elements in any random order, much as in an ENCODE or DECODE statement (see "Memory-to-Memory Data Conversion").

In READ DISC statements, words are read into the items defined by the list k, starting from the disc location defined by s. Reading is from the appropriate device.

With WRITE DISC statements, the binary word values of the list items are written on the appropriate device, starting at the location defined by s.

The value of s may be considered to be an address relative to the start of the user's file.

Auxiliary Input/Output Statements

The following set of statements enables the programmer to manipulate magnetic tapes and sequential disc 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 second 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.

Note: All declaration statements discussed in this chapter are subject to the rules for statement placement and order given at the end of the chapter.

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

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 frequency, the next leftmost dimension varies with the next highest frequency, 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 six type declarations:[t]

INTEGER
REAL
COMPLEX
LOGICAL
DOUBLE PRECISION
DOUBLE COMPLEX

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

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	COMPLEX
REAL	LOGICAL
DOUBLE PRECISION	DOUBLE COMPLEX

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 identifiers to have the types indicated:

LEVEL3 - logical

LEVEL - real

KAPPA - integer

POROUS - logical

PERCNT - real

X - real

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 a blank COMMON that contains 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 chapter).

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

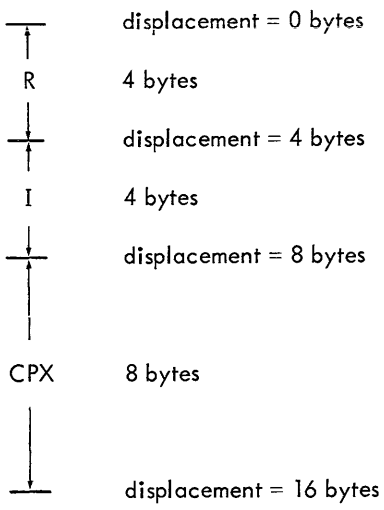
Incorrect referencing of COMMON data will terminate execution. To ensure correct referencing of data, COMMON blocks must be constructed so that the displacement of each variable in the block is an integral multiple of the reference number associated with the variable (displacement is the number of bytes 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 FLAG 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.

Example:

```
EXTERNAL F
```

Library functions (see Table 8) may not appear as arguments to a subprogram. If the name of a library function (e.g., SIN) appears in an EXTERNAL or explicit type statement, it must refer to a variable or a user-supplied subprogram.

BLOCK DATA Subprograms

FLAG 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.

BLOCK DATA subprograms must appear before the main program and all other subprograms.

Within a BLOCK DATA subprogram, initialization of labeled COMMON variables must be accomplished by one or more DATA statements; type statements may not be used for initialization.

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. Hexadecimal constants may also be used. 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 "Storage Allocation Statements" earlier in this chapter). 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. Numeric and logical constants may not be used for more than one variable list item; one literal constant may initialize successive list items or successive elements of an array appearing as a list item.

5. Hexadecimal constants may be used to initialize any type of variable. The form of a hexadecimal constant is the character Z followed by from 1 to 32 hexadecimal digits. These digits are

   0 1 2 3 4 5 6 7 8 9 A B C D E F

   As an example, the hexadecimal constant ZB0D represents the bit string 1011000011101.

   The maximum number of digits allowed in a hexadecimal constant depends on the type of variable being initialized. The following list shows the maximum number of digits for each variable type:

Type of Variable	Maximum number of Hexadecimal Digits
LOGICAL	8
INTEGER	8
REAL	8
REAL*8	16
COMPLEX	16
COMPLEX*16	32

If the number of digits is greater than the maximum, the leftmost hexadecimal digits are truncated; if the number of digits is less than the maximum, hexadecimal zeros are supplied to the left.

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.

Placement and Order of Declaration Statements

The following rules govern the placement and order of appearance of declaration statements within a main program or subprogram when using FLAG.

1. All declaration statements must appear prior to the appearance of the first executable statement within a program.

2. Declaration statements (if present) should appear in the following order within a program:

   subprogram declaration statement

   IMPLICIT statement

   type statements

   DIMENSION statements

   COMMON statements

   EQUIVALENCE statements

   EXTERNAL statements

   DATA statements

   NAMELIST statements

   Failure to follow this order may result in one or more compiler diagnostic messages.

3. Identifiers that appear both in type statements and in EQUIVALENCE statements may not have initial data values specified within the type statement; they may be initialized by one or more subsequent DATA statements.

4. Identifiers appearing in type statements within a BLOCK DATA subprogram may not have values for data-initialization specified within the type statement; all identifiers within a BLOCK DATA subprogram must be initialized by means of one or more DATA statements (or left uninitialized).

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 FLAG 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

Subroutines

SUBROUTINE subroutines

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 FLAG. These are described at the end of this chapter.

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:[t]

INTEGER	COMPLEX
REAL	LOGICAL
DOUBLE PRECISION	DOUBLE COMPLEX

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.

[t] See also "Optional 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 returned 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.

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

Table 8, below, indicates the permissible kinds of correspondence between an argument and a dummy.

Table 8. Permissible Correspondences Between Arguments and Dummies

Argument	Dummy			
	scalar	array	subprogram	asterisk
scalar or array element	yes	yes[†]	no	no
expression	yes	no	no	no
statement label	no	no	no	yes
array name	yes[†]	yes	no	no
literal constant	yes[†]	yes	no	no
subprogram name	no	no	yes	no

[†]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) = 4HIP␣M

M(3) = 4HORRI

M(4) = 4HS␣␣␣

M(5) is undefined and should not be referenced

where ␣ 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.

When running in the "debug" mode (see Chapter 9) the size of a variably dimensioned array is calculated each time the subprogram is entered, and subscripted elements of the array are checked to make sure that the subscript is in the range of the array. If a one-dimensional dummy array is dimensioned with size 1 no subscript range checking is done.

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
```

(The programmer must provide the functions SIN, DSIN, SQRT, and DSQRT.)

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

FLAG 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 9.

Table 9 lists the function subprograms provided by FLAG. 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 9, the following subprograms are supplied in the FLAG library:

EXIT

Form:

```
CALL EXIT
```

The effect is identical to that of the STOP statement.

Table 9. Intrinsic and Basic External Functions

Intrinsic Names	Number of Arguments	Type of Argument	Type of Result	Definition of Function
ABS	1	Real	Real	Absolute value. For complex, see CABS.
ACOS	1	Real	Real	Arc cosine in radians. For complex, see CACOS.
AIMAG	1	Complex	Real	Imaginary part of argument (zero if not complex) expressed as a real value.
AINT	1	Real	Real	Integer part of argument (fractional part truncated).
ALOG	1	Real	Real	Natural logarithm (base e).
ALOG10	1	Real	Real	Common logarithm (base 10).
AMAX1	N ≥ 2	Real	Real	Maximum value. All arguments are converted to and compared as real values.
AMAX0	N ≥ 2	Integer	Real	Maximum value. All arguments are converted to and compared as integer values.
AMIN1	N ≥ 2	Real	Real	Minimum value. All arguments are converted to and compared as real values.
AMIN0	N ≥ 2	Integer	Real	Minimum value. All arguments are converted to and compared as integer values.
AMOD	2	Real	Real	Arg_1 (mod arg_2). Evaluated as $arg_1 - arg_2 * AINT(arg_1/arg_2)$ i.e., the sign is the same as arg_1. Function undefined if $arg_2 = 0$.
ASIN	1	Real	Real	Arc sine in radians. For complex, see CASIN.
{ATAN / ATAN2}	1, 2	Real	Real	Arctangent in radians. Arg_1 = ordinate (y), arg_2 = abscissa (x). If arg_2 is not present, assumed 1. Result (R) is arctangent of arg_1/arg_2 quadrant allocated in the range $-\pi < R \leq \pi$. ATAN(0, 0) = 0. For complex, see CATAN.
CABS	1	Complex	Real	Complex absolute value (i.e., modulus). $CABS(x + iy) = \sqrt{x^2 + y^2}$
CACOS	1	Complex	Complex	Complex arc cosine. CACOS(Z) $= -i \cdot CLOG(Z + CSQRT(Z^2 - 1))$
CASIN	1	Complex	Complex	Complex arc sine. CASIN(Z) $= -i \cdot CLOG(i \cdot Z + CSQRT(1 - Z^2))$
CATAN	1	Complex	Complex	Complex arctangent. CATAN(Z) $= u + iv = -\frac{i}{2}$ (CLOG(1 + iZ) - CLOG(1 - iZ)), allocated such that $-\pi < u \leq \pi$.
CCOS	1	Complex	Complex	Complex cosine. CCOS(Z) $= (e^{iZ} + e^{-iZ})/2$.

Table 9. Intrinsic and Basic External Functions (cont.)

Intrinsic Names	Number of Arguments	Type of Argument	Type of Result	Definition of Function
CCOSH	1	Complex	Complex	Complex hyperbolic cosine. CCOSH(Z) $= (e^{Z} + e^{-Z})/2$.
CDABS	1	Complex*16	Real*8	Double complex absolute value (modulus). See CABS.
CDACOS	1	Complex*16	Complex*16	Double complex arc cosine. See CACOS.
CDASIN	1	Complex*16	Complex*16	Double complex arc sine. See CASIN.
CDATAN	1	Complex*16	Complex*16	Double complex arc tangent. See CATAN.
CDBLE	1	Complex	Complex*16	Converts complex to double complex.
CDCOS	1	Complex*16	Complex*16	Double complex cosine. See CCOS.
CDCOSH	1	Complex*16	Complex*16	Double complex hyperbolic cosine. See CCOSH.
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.
CDSINH	1	Complex*16	Complex*16	Double complex hyperbolic sine. See CSINH.
CDSQRT	1	Complex*16	Complex*16	Double complex square root. See CSQRT.
CDTAN	1	Complex*16	Complex*16	Double complex tangent. See CTAN.
CDTANH	1	Complex*16	Complex*16	Double complex hyperbolic tangent. See CTANH.
CEXP	1	Complex	Complex	Complex exponential (e ** arg). CEXP(x + iy) = EXP(x) · (COS(y) + i · SIN(y)).
CLOG	1	Complex	Complex	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	Complex	Converts two noncomplex numbers to a complex number. CMPLX(x,y) = x + iy.
CONJG	1	Complex	Complex	Complex conjugate. CONJG(x + iy) = x - iy.
COS	1	Real	Real	Cosine of angle in radians. For complex, see CCOS.
COSH	1	Real	Real	Hyperbolic cosine. For complex, see CCOSH.
CSIN	1	Complex	Complex	Complex sine. CSIN(Z) $= (e^{iZ} - e^{-iZ})/(2i)$.

Table 9. Intrinsic and Basic External Functions (cont.)

Intrinsic Names	Number of Arguments	Type of Argument	Type of Result	Definition of Function
CSINH	1	Complex	Complex	Complex hyperbolic sine. CSINH(Z) $= (e^{Z} - e^{-Z})/2.$
CSNGL	1	Complex*16	Complex	Converts double complex to complex.
CSQRT	1	Complex	Complex	Complex square root. CSQRT(z) $= u + iv = e^{(\ln Z)/2}$, allocated such that $u \geq 0$.
CTAN	1	Complex	Complex	Complex tangent. CTAN(Z) $= \text{CSIN}(Z)/\text{CCOS}(Z)$ $= (e^{iZ} - e^{-iZ})/i(e^{iZ} + e^{-iZ}).$
CTANH	1	Complex	Complex	Complex hyperbolic tangent. CTANH(Z) = CSINH(Z)/CCOSH(Z) $= (e^{Z} - e^{-Z})/e^{Z} + e^{-Z}).$
DABS	1	Real*8	Real*8	Double precision absolute value.
DACOS	1	Real*8	Real*8	Double precision arc cosine in radians.
DASIN	1	Real*8	Real*8	Double precision arc sine in radians.
{DATAN / DATAN2}	1, 2	Real*8	Real*8	Double precision arctangent in radians. See ATAN.
DBLE	1	Real	Real*8	Argument converted to a value with double precision.
DCMPLX	2	Real*8	Complex*16	Converts two noncomplex 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 consine of angle in radians.
DCOSH	1	Real*8	Real*8	Double precision hyperbolic cosine.
DDIM	2	Real*8	Real*8	Double precision positive difference. See DIM.
DEXP	1	Real*8	Real*8	Double precision exponential (e ** arg).
DFLOAT	1	Integer	Real*8	Argument converted to double precision. Same as DBLE, but generally used with integer arguments.
DIM	2	Real	Real	Positive difference. DIM(x,y) $= x - \min(x,y)$.
DIMAG	1	Complex*16	Real*8	Imaginary part of a double complex argument, expressed as a double precision value.
DINT	1	Real*8	Real*8	Integer part of the argument expressed as a double precision value.
DLOG	1	Real*8	Real*8	Double precision natural logarithm (base e).

Table 9. Intrinsic and Basic External Functions (cont.)

Intrinsic Names	Number of Arguments	Type of Argument	Type of Result	Definition of Function
DLOG10	1	Real*8	Real*8	Double precision common logarithm (base 10).
DMAX1	N ≥ 2	Real*8	Real*8	Double precision maximum value. All arguments are converted to and compared as double precision values.
DMIN1	N ≥ 2	Real*8	Real*8	Double precision minimum value. All arguments are converted to and compared as double precision values.
DMOD	2	Real*8	Real*8	Double precision arg_1 (mod arg_2). See AMOD.
DREAL	1	Complex*16	Real*8	Real part of a double complex argument, expressed as a double precision value.
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.
DSINH	1	Real*8	Real*8	Double precision hyperbolic sine.
DSQRT	1	Real*8	Real*8	Double precision square root (positive value).
DTAN	1	Real*8	Real*8	Double precision tangent.
DTANH	1	Real*8	Real*8	Double precision hyperbolic tangent.
EXP	1	Real	Real	Exponential (e ** arg). For complex, see CEXP.
FLOAT	1	Integer	Real	Argument converted to a real value.
IABS	1	Integer	Integer	Integer absolute value.
IAND	2	Integer	Integer	Logical AND (extract).
ICOMPL	1	Integer	Integer	Logical NOT (1's complement). Same as INOT.
IDIM	2	Integer	Integer	Integer positive difference. IDIM(j,k) = j - MIN(j,k).
IDINT	1	Real*8	Integer	Argument converted to an integer value.
IEOR	N ≥ 2	Integer	Integer	Logical EOR (exclusive OR).
IEXCLR	N ≥ 2	Integer	Integer	Logical EOR (exclusive OR). Same as IEOR.
INOT	1	Integer	Integer	Logical NOT (1's complement).
{INT / IFIX}	1	Real	Integer	Argument converted to an integer value.

Table 9. Intrinsic and Basic External Functions (cont.)

Intrinsic Names	Number of Arguments	Type of Argument	Type of Result	Definition of Function
IOR	2	Integer	Integer	Logical OR (merge).
ISA	2	Integer	Integer	Integer shift arithmetic. Arg_1 is shifted left arithmetically the number of bits specified in arg_2. If arg_2 is negative, the shift is to the right.
ISC	2	Integer	Integer	Integer shift circular. Arg_1 is shifted left circularly the number of bits specified in arg_2. If arg_2 is negative, the shift is to the right.
ISIGN	2	Integer	Integer	Integer magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive. Arg_2 is not converted to integer.
ISL	2	Integer	Integer	Integer shift logical. Arg_1 is shifted left logically the number of bits specified in arg_2. If arg_2 is negative, the shift is to the right.
MAX0	N ≥ 2	Integer	Integer	Integer maximum value.
MAX1	N ≥ 2	Real	Integer	Integer maximum value.
MIN0	N ≥ 2	Integer	Integer	Integer minimum value. All arguments are converted to and compared in integer.
MIN1	N ≥ 2	Real	Integer	Integer minimum value. All arguments are converted to and compared in double precision.
MOD	2	Integer	Integer	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	Real	Real part of a complex number.
SIGN	2	Real	Real	Magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
SIN	1	Real	Real	Sine of angle in radians. For complex, see CSIN.
SINH	1	Real	Real	Hyperbolic cosine. For complex, see CSINH.
SNGL	1	Real*8	Real	Argument converted to a value with real (single) precision.
SQRT	1	Real	Real	Square root (positive value). For complex, see CSQRT.
TAN	1	Real	Real	Tangent of angle in radians. For complex, see CTAN
TANH	1	Real	Real	Hyperbolic tangent. For complex, see CTANH.

SLITET – Sense Light Test

Form:

CALL SLITET (n,v)

where

n is an integer constant or scalar variable 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 constant or scalar variable ($0 \leq n \leq 4$).

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.

9. OPERATIONS

FLAG operates under control of the Sigma 5/7 Batch Processing Monitor (BPM). Preparing a FLAG job for compilation or combined compilation and execution is a simple procedure requiring the preparation of a few control cards. This is one of FLAG's most attractive features, together with its facility for rapid compilation and execution of programs.

The user has a number of convenient processing options at his disposal, all of which can be controlled by option codes on the FLAG control card. Option codes are explained later in this chapter in Table 10.

Running a FLAG Job

Figures 3 and 4 later in this chapter show sample deck setups for compiling and executing FLAG jobs. The JOB, ASSIGN, EOD, and FIN cards shown in the examples are standard BPM control cards. Many installations using BPM arrange for the computer operator to insert these control cards in the FLAG deck, the programmer supplying only the FLAG control card required for his job.

The ASSIGN cards shown in the examples are necessary only if the programmer requires file or I/O device assignments different from the standard ones provided at his installation. The source program decks and EOD cards would be omitted if M:SI were not assigned to the card reader. That is, the program decks together with their appropriate end-of-data indications could be read from magnetic tape or disc. Detailed information on all control cards, except the FLAG card, is available in the Sigma 5/7 Batch Processing Monitor Reference Manual (SDS 90 09 54).

The FLAG Control Card

Every FLAG job must be preceded by a FLAG control card. Its format is

```
!FLAG [option1, option2, . . ., optionn]
```

The exclamation mark must be placed in column 1. The FLAG control command is usually begun in column 2, though it may begin in any column after the ! character. The $option_i$ are option codes that control processing and execution of the program. Option codes are not required; if none are specified, FLAG will perform certain operations by default. If option codes are given, they must be separated by commas, and the first code must be preceded by at least one blank column. (The brackets around the option list shown above must not be entered on the card: they indicate only that the list of options is not required.) The option codes are given in Table 10. In Table 10, the notation DEFAULT indicates which options are in effect unless their complementary options have been selected.

When the BJ option code is specified on the ! FLAG card, FLAG enters the batch job mode. In this mode, FLAG will successively compile and execute any number of separate FLAG programs, or "subjobs". Use of this option substantially reduces the processing time required for each program in the job stream.

Once FLAG has read the BJ option code on the ! FLAG card it expects to find a :FLAG card immediately preceding each subjob. Figure 5 illustrates how the :FLAG card is used in batch processing. Format of the :FLAG card is

```
:FLAG [(account number, name)] [,option1,option2, . . .,optionn]
```

On this card the characters :FLAG must appear in columns 1-5. The user's account number and name are required on every :FLAG card if the AC option has been specified on the ! FLAG card; otherwise they are optional. All standard ! FLAG option codes are valid on a :FLAG card except for NOBJ, AC, and NOAC. (The brackets around account number, name, and the list of option codes indicate only that these items are not required; the brackets are not actually entered on the card.)

Any I/O unit number assignments in effect when the batch job run is started remain in effect for all subjobs.

The source program deck for each subjob must be separated from its data by an :EOD card. (An !EOD card will also work but :EOD is preferred.) If no data is present, the :EOD card is still required to terminate the source program deck.

The batch job stream is terminated by the first ! control card encountered (other than :EOD).

Table 10. FLAG Option Codes

Option Code	Meaning
DB	The program is compiled and executed in "debug" mode.
NODB	The program is not compiled in "debug" mode. (DEFAULT) Note that use of the DB option will cause substantially more machine instructions to be generated for the program (typically 30 to 40% more), and that some programs that are too large to run with "debug" may be able to run without it.
GO	The program is executed when compilation is finished, whether or not errors have been detected during compilation.
NOGO	When compilation is finished, the program is not executed. This option allows syntax checking of the source program without execution. Note that if neither GO nor NOGO are specified, the compiled program will still be executed unless one or more serious errors have been detected during compilation.
LS	A printed listing of the FORTRAN statements in the source program is produced. (DEFAULT)
NOLS	No listing of the source statements is produced.
BC	FLAG will compile a series of source programs, each followed by a single end-of-data (EOD) indication, until two successive end-of-data indications are encountered. Then the series will be executed, if appropriate. This option is mainly intended for use where a main program and subprograms are to be compiled as a unit but have been stored on magnetic tape, and therefore each program is followed by an EOD record. The BC option code alerts FLAG to this condition and prevents it from assuming "end-of-program" when a single EOD is encountered.
NOBC	FLAG will terminate compilation upon encountering an !EOD card or some other single end-of-data indicator. (DEFAULT)
LO	A machine-language listing of the instructions generated by FLAG is produced in a format similar to a Meta-Symbol listing.
NOLO	No machine-language listing is produced. (DEFAULT)
AD	All REAL variables, constants, and functions are implicitly REAL*8; all COMPLEX variables, constants, and functions are implicitly COMPLEX*16. This option is useful for analyzing the improvement in accuracy that results from double precision calculations.
NOAD	No "automatic double precision" is invoked. (DEFAULT)
CX	Source card-images containing an X in column 1, will have the X replaced by a space, and will be compiled by FLAG.
NOCX	Source card-images containing an X in column 1 will be treated as comment cards by FLAG. (DEFAULT)
Mn	(Where n is a digit 1 through 9) FLAG divides total available memory into two segments. One segment consists of the noninitialized variables that are used in the program. The other segment contains the machine instructions generated for the program and also any variables that were initialized in a DATA statement. The n value specifies how many tenths of the available memory are to be used for the noninitialized data area. For example, M8 specifies that 8/10 of memory is to be used for noninitialized variables and 2/10 is to be used for the program code and initialized variables. This option need not be

Table 10. FLAG Option Codes (cont.)

Option Code	Meaning
Mn (cont.)	specified unless FLAG indicates by one of the following error messages that a memory size allocation problem has occurred: Error Message — Action to Take ARRAYS TOO LARGE — increase n PROGRAM TOO LARGE — decrease n DICTIONARY OVERFLOW — increase n The default option is M7.
BJ	Enter "batch job" mode. A subjob beginning with a :FLAG card must immediately follow the !FLAG control card. All subsequent subjobs must also begin with :FLAG cards.
NOBJ	Run in standard (nonbatch) mode. (DEFAULT)
AC	Punch separate accounting records for each subjob in the batch. The AC option, when used, should appear on the !FLAG control card that initiates the batch run.
NOAC	No accounting records will be punched for subjobs. (DEFAULT)
TL=sss	Set time limit for current job or subjob, where sss is the number of seconds that the job may run. This time includes compilation and execution.
PL=ppp	Set page limit for current job or subjob, where ppp is the number of output pages that will be allowed. Note that if either the TL or PL option appears on a !FLAG control card that also has the BJ option, the value specified is used as the default limit for each of the subjobs that follow. Furthermore, the limit specified on a !FLAG control card becomes the maximum limit that may be specified on a subjob's :FLAG control card.
PS=nnn	Set program size for current job or subjob, where nnnn is the number of words of memory that may be used to hold the program and any variables that were initialized in a DATA statement. The remainder of available memory will be used to hold the noninitialized variables. The PS option is similar to the Mn option but allows for more accurate allocation of memory size. The actual amount of memory used by the program and initialized variables is given at the end of the source program listing. When attempting to run very large programs, it is sometimes a good idea to make the first compilation using the NOGO and M9 options. When NOGO is specified no machine instructions are stored into memory, hence substantially less memory is needed for the program and initialized variable area. The actual program size, which is listed at the end of the NOGO compilation, is correct and is the same as the size of the program when NOGO is not specified. If the actual total size is less than the available total size, it is possible to run the program, and the program should be submitted again, this time with the PS option set equal to the indicated size of the program and initialized variable area.

Job Setup Examples

Figure 3 shows the deck setup required for compiling and executing a single program. The !JOB card signals the beginning of a new job to BPM, and specifies that the job is to be run under account number 1234, the user is SHERROD, and the job has priority 1.

As previously explained, the !ASSIGN card (with assignment codes) would be present only if the programmer required nonstandard assignments.

The !FLAG card summons the FLAG compiler to begin compilation of the source program. The option codes following the FLAG command are explained in Table 10. Briefly, the codes shown in the example request the program to be compiled and executed in "debug" mode (DB), execute the program when compilation is finished regardless of errors (GO), produce a listing of the source program statements (LS), and produce a listing of the machine-language statements generated by the compiler (LO).

The !FLAG card is followed by the source program deck, in turn followed by an !EOD card which indicates the end of the deck to the compiler. If the source program did not require a data deck, the !EOD card could be omitted and end-of-program could be indicated by some other terminator such as a !FIN card or a new !JOB card. The source deck and its terminating !EOD card would be omitted if M:SI were not assigned to the card reader.

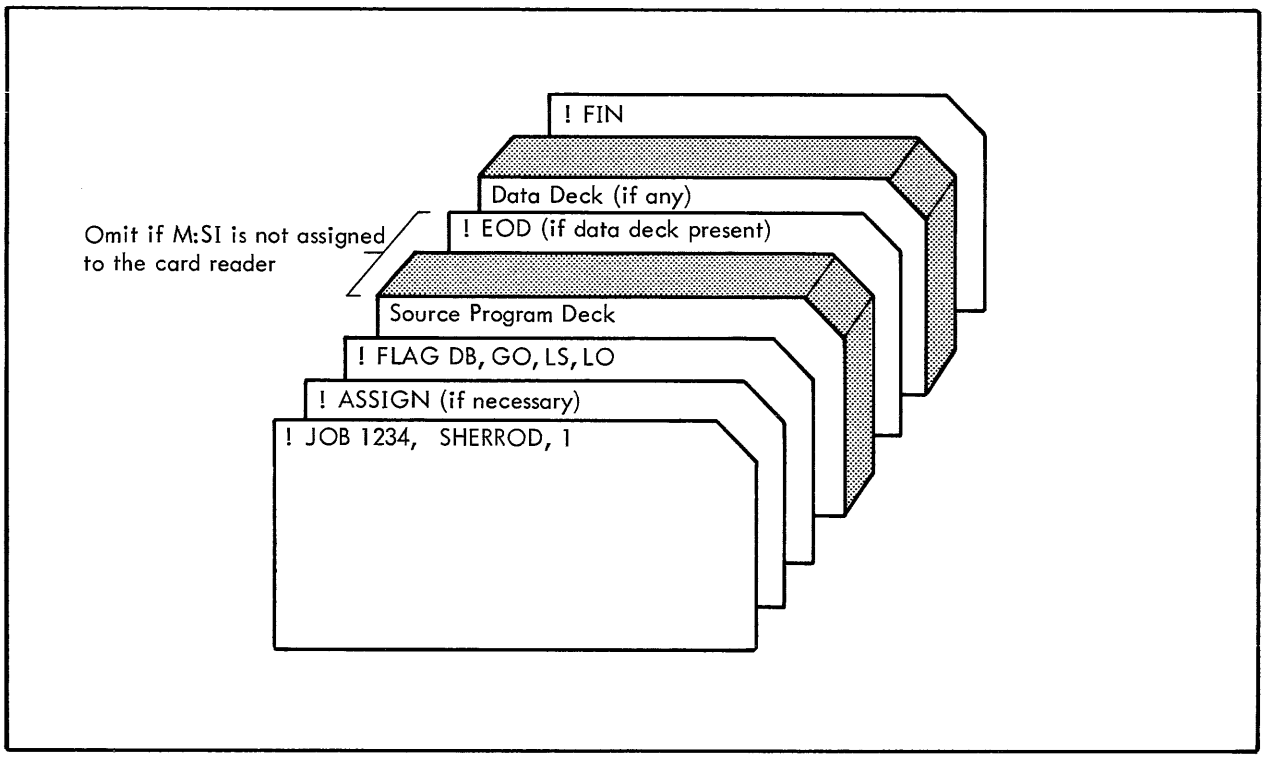


Figure 3. FLAG Job Setup – Single Program

The source program deck can consist of either a single program or a main program followed by a number of subprograms. Either will compile and execute correctly. For the latter case, it is not necessary to use the BC option or separate the programs and subprograms with EOD cards as long as M:SI is assigned to the card reader.

Figure 4 illustrates the deck setup for compiling and executing a series of independent programs in the batch job mode. The functions of the !JOB, !ASSIGN, and !FLAG cards are the same as explained for Figure 3.

The BJ option on the !FLAG card specifies that FLAG is to enter the batch job mode. Each of the following programs must then be preceded by a :FLAG card containing option codes, if appropriate, for the program. Note that each program deck is followed by an :EOD card, even source program number 2, which does not have a data deck. The series can continue indefinitly until terminated by a !FIN card.

FLAG Debug Mode

If the user elects compilation and execution in "debug" mode (see FLAG DB option), the FLAG compiler will generate extra instructions in the compiled program so that program errors that cannot be detected during compilation will be detected during program execution. This enables the user to detect errors in program logic that otherwise might go undetected or cause unexplainable program failures. The following errors are reported by the "debug" option:

1. Subscripts having values that are negative, zero, or larger than the specified dimension size.

2. Inconsistencies in type or number of arguments passed to subprograms.

3. Arithmetic underflow, overflow, and division by zero. (If the intrinsic subprogram DVCHK or OVERFL has been referenced, these conditions are not considered errors, and no debug error message will be produced.)

Additionally, when any of the errors mentioned above is detected, or when an error is detected within an intrinsic subprogram or input/output routine, debug mode compilation will cause the program name and line number of the FORTRAN statement being executed to be listed, along with a listing of all subprogram calls in effect at the time of the error.

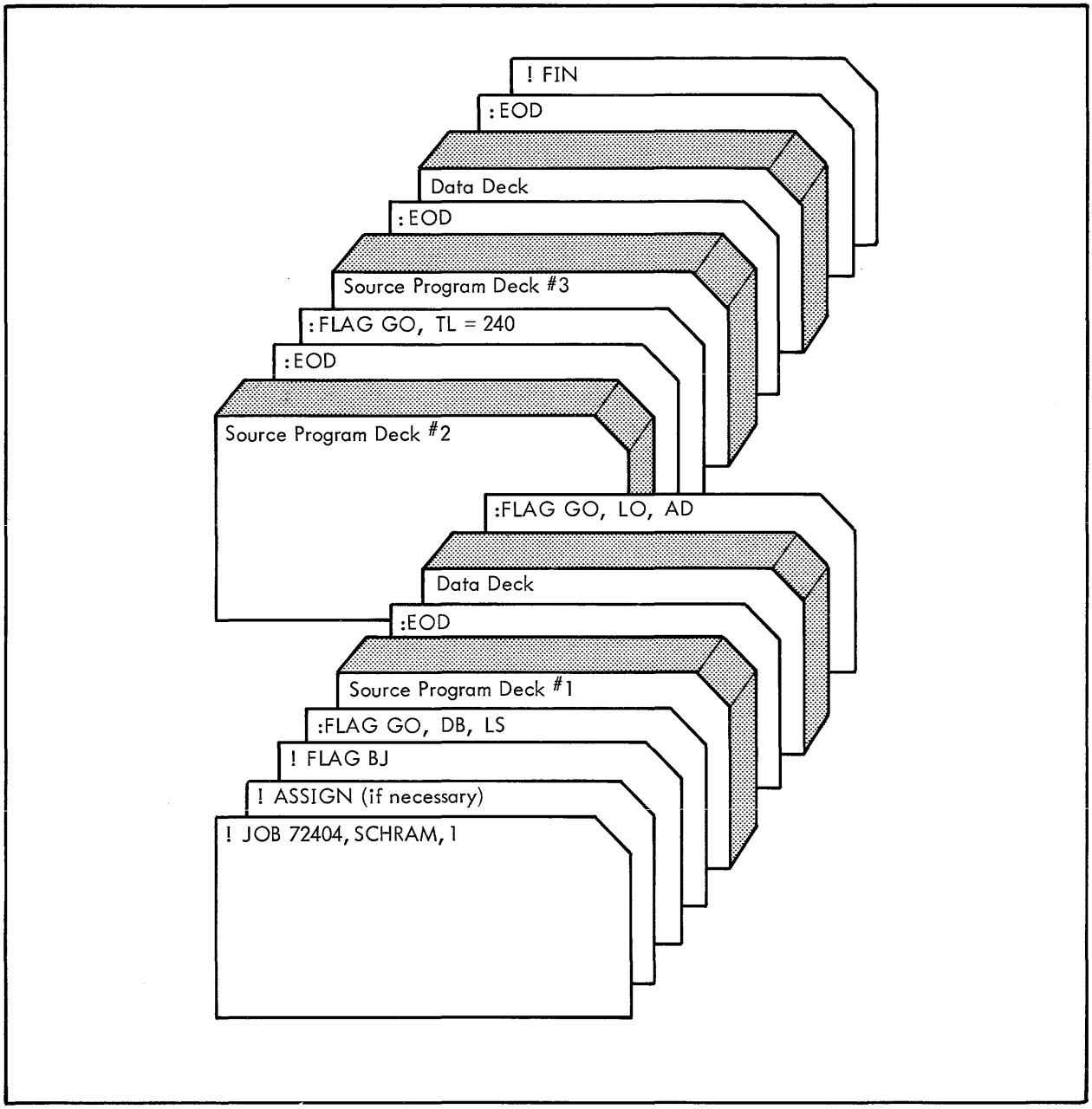


Figure 4. FLAG Job Setup – Multiple Programs in Batch Processing Mode

APPENDIX A. FLAG STATEMENTS

Statement	Executable	Nonexecutable	Page
ASSIGN	X		19
Assignment	X		16
BACKSPACE	X		64
BLOCK DATA		X	76
CALL	X		21
COMMON		X	70
COMPLEX		X	68
CONTINUE	X		25
DATA		X	77
DECODE	X		60
DIMENSION		X	67
DO	X		22
DOUBLE COMPLEX		X	68
DOUBLE PRECISION		X	68
ENCODE	X		60
END		X	26
END FILE	X		64
EQUIVALENCE		X	73
EXTERNAL		X	76
FORMAT		X	37
FUNCTION		X	81
GO TO	X		18
IF	X		20
IMPLICIT		X	67
INPUT	X		34
INTEGER		X	68
LOGICAL		X	68
NAMELIST		X	32

Statement	Executable	Nonexecutable	Page
OUTPUT	X		33
PAUSE	X		25
PRINT	X		30, 31
PUNCH	X		30
READ	X		29-32
READ DISC	X		63
REAL		X	68
RETURN	X		22
REWIND	X		64
STOP	X		25
SUBROUTINE		X	82
Statement Function Definition		X	80
WRITE	X		29, 31
WRITE DISC	X		63

APPENDIX B. DIAGNOSTIC MESSAGES

Listed below are all the diagnostic messages that FLAG can produce during compilation or execution of programs. The messages are printed on the device assigned as M:LO, and are interspersed with the symbolic listing of the source statements. Many of the messages have a name or a statement number either inserted in the message or following it to indicate the source of the error. Some messages are merely warnings to the programmer and will not cause the job to be aborted. Other messages are notifications of serious error; these will cause the job to be aborted once the compilation is completed. During compilation of a source program, messages that are only warnings are not printed unless compilation is being performed in the debug mode (DB option).

ABORTED INSTRUCTION = X'dddddddd'
ACCOUNT # AND NAME MISSING
ACTUAL PROGRAM SIZE:
ADDRESS OF ABORTED INSTRUCTION = X'ddddd'
ARGUMENT NUMBER
ARITH OVRFL:
ARITHMETIC ASSIGNMENT STATEMENT
ARRAYS TOO LARGE
'ASSIGN' MISSPELLED
ASSIGNMENT MEMORY SIZE:

'BACKSPACE' MISSPELLED
BAD HOLLERITH COUNT
BAD REPEAT COUNT
BLANK CARD IN PROGRAM
'BLOCK DATA' NOT FIRST PROGRAM
'BLOCK DATA' NOT FIRST STMNT

CANNOT REACH STMNT : #
CHANGED SIZE OF BLOCK
CHANGED VALUE OF DO INDEX:
COMMON AFTER DATA STMNT
COMMON AFTER EQUIVALENCE
COMMON EXTENDED BACKWARD BY xxxxx
'COMMON' MISSPELLED
'COMPLEX' MISSPELLED
COMPLEX NO. RAISED TO NON-INTEGER POWER
'CONTINUE' MISSPELLED

DEGENERATE EQUIVALENCE GROUP
DICTIONARY OVERFLOW
'DIMENSION' MISSPELLED
DIMENSIONED VARIABLE HAS NO SUBSCRIPT:
DIV BY ZERO
DO ENDS ON PREVIOUS STMNT
DO INTERSECTS ANOTHER DO
DO'S NESTED TOO DEEPLY
'DOUBLE COMPLEX' MISSPELLED
'DOUBLE PRECISION' MISSPELLED
DUMMY IN EQUIVALENCE
DUPLICATE STMNT # :
DUPLICATE SUBPROGRAM NAME

EARLIER STMNT TRANSFERS TO FORMAT.
EFFECTIVE ADDRESS = X'ddddd'
x ENCOUNTERED INSTEAD OF NAME
END AND ERR OPTIONS NOT ALLOWED IN WRITE STMNT
'END FILE' MISSPELLED
EQUAL SIGN MISSING
EQUIVALENCE AFTER DATA INITIALIZATION
EQUIVALENCE CONTRADICTION.

'EQUIVALENCE' MISSPELLED
ERROR IN ABS READ FROM DO
ERROR IN IMPLIED DO
ERROR TO LEFT OF EQUAL SIGN.
ERRORED AT LINE #
'xxxx' EXCEEDS 5 DIGITS
'xxxx' EXCEEDS 6 CHARS.
EXCESS INFORMATION IGNORED
EXECUTABLE STMNT IN BLOCK DATA
EXPRESSION MUST BE INTEGER OR REAL
'EXTERNAL' MISSPELLED
EXTRA COMMA
EXTRA IMPLICIT IGNORED

FLAG VERSION 34
FORMAT ARRAY NOT DIMENSIONED:
'FORMAT' MISSPELLED.
FORMAT MUST HAVE STMNT # .
FORMAT NOT USED: #
FUNCTION HAS NO DUMMIES
FUNCTION HAS TOO MANY ARGUMENTS:
'FUNCTION' MISSPELLED
'FUNCTION' STMNT NOT FIRST STMNT

ILLEGAL ARGUMENT TYPE IN
ILLEGAL EQUIVALENCE OF xxxxxxxx TO xxxxx
ILLEGAL EXPONENTIATION POWER
ILLEGAL SUBSCRIPT VALUE
**ILLEGAL TRAP ... JOB ABORTED
ILLEGAL TYPE WITH RELATIONAL
ILLEGAL USE OF '.NOT.'
ILLEGAL USE OF COMMA
ILLEGAL USE OF DIMENSIONED VARIABLE:
ILLOGICAL EXPRESSION
'IMPLICIT' MISPLACED
'IMPLICIT' MISSPELLED
IMPROPER STMNT WITH LOGICAL IF
INCOMPLETE DATA.
'INPUT' MISSPELLED
'INTEGER' MISSPELLED
INTEGER TOO BIG
INVALID ARGUMENT
INVALID ARGUMENT TO xxxx
INVALID COMPLEX CONSTANT.
INVALID DATA VALUE.
INVALID DELIMITER
INVALID DIMENSION SIZE
INVALID EXPONENT
INVALID EXPRESSION
INVALID FLAG-CARD OPTION ... JOB ABORTED.
INVALID FORMAT SYNTAX
xx INVALID IN CALL TO xxxx
INVALID LOGICAL OPERATOR
INVALID MESSAGE IN PAUSE STMNT
INVALID MODE.
INVALID 2ND USE OF xxxxx
INVALID SIZE SPECIFICATION
INVALID STMNT #
INVALID SYNTAX
INVALID SYNTAX IN I/O LIST
INVALID TERMINAL STMNT OF DO LOOP
I/O DEVICE # MISSING
IO DEVICE # MUST BE UNSIGNED INTEGER

'LOGICAL' MISSPELLED
LOGICAL MODE WITH ARITHMETIC OPERATOR

**MAX PAGES OUT ... JOB ABORTED
**MAX TIME ... JOB ABORTED
MISALIGNED DOUBLE-WORD VARIABLE:
MISMATCHED PARENS
MISPLACED DECLARATIVE STMNT.
MISPLACED OPERATOR
MISSING COMMA
MISSING DELIMITERS
MISSING END STATEMENT
MISSING :EOD CARD ... THAT'S A NO NO ... JOB ABORTED
MISSING FORMAT: #
MISSING OPEN PAREN
MISSING OPERATOR
MISSING OR INVALID INDEX VARIABLE
MISSING OR INVALID INITIAL DO VALUE
MISSING SIZE SEPCIFICATION
MISSING SLASH
MISSING STMNT : #
MISSING SUBPROGRAM:
MIXED LOGICAL & ARITH EXPRESSIONS
MIXED PRECISION COMPLEX CONSTANT
MORE THAN 1 MAIN PROGRAM
MUST BE INTEGER:
MUST BE UNSIGNED INTEGER CONSTANT

NAMELIST CONTAINS DUMMY VARIABLE:
'NAMELIST' MISSPELLED
NAME PREVIOUSLY USED AS FUNCTION:
NO DIMENSIONING INFORMATION
NO MAIN PROGRAM
NON-ALPHABETIC ORDER.
NON-DIMENSIONED VARIABLE HAS SUBSCRIPT:
NON-DUMMY HAS VARIABLE DIMENSION:
NON-INITIALIZED DATA =
NUMBER EXCEEDS LIMITS
NUMBER OF FATAL ERRORS DETECTED =
NUMBER OF NAMES DOES NOT MATCH NUMBER OF VALUES.

1 OR MORE INVALID CHARS SKIPPED
ONLY DIGITS MAY FOLLOW 'STOP'
ONLY 1 ARGUMENT TO xxxx
OPERATOR FOLLOWS OPERATOR
'OUTPUT' MISSPELLED
OVER 7 DIMENSIONS
OVER 7 SUBSCRIPTS
OVER 19 CONTINUATION CARDS ... JOB ABORTED

PREVIOUS STMNT TRANSFERS INTO DO LOOP
PROGRAM AND INITIALIZED DATA =
PROGRAM EXECUTION NOT ATTEMPTED
PROGRAM HAS INPUT STMNT BUT NO NAME LIST STMNT
PROGRAM TOO LARGE

'RETURN' MISSPELLED
RETURN STMNT IN MAIN PROGRAM
'REWIND' MISSPELLED

STATEMENT MUST BEGIN WITH A LETTER
STMNT NUMBERS MISSING
SUBPROGRAM CALLS ITSELF:

SUBPROGRAM NOT USED:
'SUBROUTINE' MISSPELLED
'SUBROUTINE' STMNT NOT FIRST STMNT
SUBSCRIPT HAS ILLEGAL MODE
SUBSCRIPT MUST BE INTEGER CONSTANT
SUBSCRIPT OUT OF RANGE
SUBSCRIPT OUT OF RANGE, LINE #

TOO MANY I/O UNIT-NUMBER ASSIGNMENTS
TOTAL =
TRANSFER STMNT ENDS DO LOOP
TRANSFERS INTO DO LOOP AT STMNT #
TRANSFERS TO FORMAT : #
TRANSFERS TO NONEXECUTABLE STMNT : #
TRANSFERS TO SELF
TYPE ALREADY ASSIGNED:

UNDEFINED VARIABLE:
UNIMPLEMENTED SIZE IGNORED
UNIMPLEMENTED STATEMENT
UNNUMBERED CONTINUE STMNT
UNNUMBERED STMNT FOLLOWS RETURN
UNNUMBERED STMNT FOLLOWS STOP
UNNUMBERED STMNT FOLLOWS TRANSFER
UNRECOGNIZABLE STATEMENT
UNSATISFIED DO : #
UNTERMINATED QUOTE FIELD
USE CONFLICTS WITH PRIOR DECLARATION:

VALUE NOT SAME TYPE AS xxxx
VARIABLE ALREADY DIMENSIONED:
VARIABLE ALREADY IN COMMON:
VARIABLE APPEARS TWICE IN DUMMY LIST
VARIABLE DECLARED BUT NOT USED:
VARIABLE DEFINED BUT NOT USED:
VARIABLE MAY NOT BE DIMENSIONED:

WRONG NUMBER OF ARGS TO xxxx
WRONG NUMBER OF ARGUMENTS
WRONG NUMBER OF SUBSCRIPTS

INDEX

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

E

F

G

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

H

I

J

L

M

N

O

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.