SDS
SCIENTIFIC DATA SYSTEMS

SDS 900 SERIES
Fortran II

SDS 900 Series Fortran II

The use of this manual presupposes familiarity with the basic concepts of FORTRAN II programming on the part of the reader. No attempt has been made to present the essentials of FORTRAN II in a form which might be used as a fundamentals course in the method. The primary purpose of the manual is to generally describe the statements and capabilities of SDS 900 Series FORTRAN II.

SDS
SCIENTIFIC DATA SYSTEMS

FIGURE 1

SDS Fortran II Coding Form

Prepared By _____ Date _____

Problem Identification _____ Page _____ of _____

```
C  CALCULATE AND TYPE THE ROOTS OF A SET OF QUADRATIC EQUATIONS
      DIMENSION A(100),B(100),C(100)
      ACCEPT TAPE 1,N,(A(J),B(J),C(J),J=1,N)
1     FORMAT(I3/(1P3E12.4))
      DO 2 I=1,N
      DISCR=B(I)**2-4.*A(I)*C(I)
C  IF ROOTS ARE COMPLEX GO TO 3 IF NOT GO TO 4
      IF(DISCR)3,4,4
3     ROOT1RL=-B(I)/(2.*A(I))
      ROOT2RL=ROOT1RL
      ROOT1IM=SQRTF(-DISCR)/(2.*A(I))
      ROOT2IM=-ROOT1IM
      GO TO 2
4     ROOT1RL=(-B(I)+SQRTF(DISCR))/(2.*A(I))
      ROOT2RL=(-B(I)-SQRTF(DISCR))/(2.*A(I))
      ROOT1IM=0.
      ROOT2IM=0.
2     TYPE 5,A(I),B(I),C(I),ROOT1RL,ROOT1IM,ROOT2RL,ROOT2IM
5     FORMAT(3H A=1PE11.4,3H B=E11.4,3H C=E11.4/6X,7HROOT1=(
     1E11.4,4H)+I(E11.4,8H) ROOT2=(E11.4,4H)+I(E11.4,1H))
      STOP
      END
```

SDS A 144

INTRODUCTION

This manual contains the preliminary specifications of SDS 900 Series FORTRAN II which provides engineers and scientists with an efficient and easily understood means of writing programs for the SDS 900 Series computers. It consists of the SDS 900 Series FORTRAN II language and the SDS 900 Series FORTRAN II processor.

The SDS 900 Series FORTRAN II language is problem-oriented as opposed to machine-oriented. This allows a programmer to concentrate on the problem to be solved rather than the details of computer operation. Expressions are used which are quite similar to accepted mathematical notation involving the operational relationships of constants, variables, and functions. Special instructions to the computer, such as input or output commands and data specifications, use names or mnemonics easily associated with corresponding English terms.

The SDS 900 Series FORTRAN II processor takes programs written in the SDS 900 Series FORTRAN II language and produces or compiles machine language programs for execution on an SDS 900 Series computer. To minimize the expense of compiling, which is a major expense of today's computing installations, the SDS 900 Series FORTRAN II design emphasizes compiling speed. In addition, features are included for use at both run time and compile time to reduce the cost and time required for program checkout.

The SDS 900 Series FORTRAN II processor contains additional features (such as ACCEPT, TYPE) and fewer restrictions (for example, mixed expressions are permitted) than FORTRAN II processors written for other computers. These FORTRAN II processors, however, are a direct subset of the SDS 900 Series FORTRAN II processor.

The SDS 900 Series FORTRAN II processor will compile and run FORTRAN II programs written for other computers, provided only that certain reasonable restrictions are met. These restrictions are:

1. The memory capacity of the SDS 900 Series computer must be sufficient to hold the compiled program and all subroutines required at run time.
2. All peripheral equipment (such as magnetic tapes) called for in the program must be attached to the SDS 900 Series computer.
3. Integer quantities are limited to 8,388,607 and floating point precision is limited to approximately twelve decimal digits.
4. The program must be a legal FORTRAN II program, i.e., one that does not use the veiled characteristics of a particular compiler-computer pair to achieve a result in variance with, or not covered by, the currently accepted definition of FORTRAN II statements and programs as outlined in this manual.

The SDS 900 Series FORTRAN II processor has been designed to take advantage of the SHARE library.

SDS 900 SERIES FORTRAN II PROGRAMS

Figure 1 illustrates an SDS 900 Series FORTRAN II program written on a standard SDS 900 Series FORTRAN II coding form. The program consists of a sequence of statements. Each statement is written on a separate line; however, if a statement is too long to fit one line, as many as nine continuation lines can be used. Continuation lines are specified by a non-zero character in column 6 of the coding form.

Statement numbers are written in columns 1 through 5. These numbers permit cross reference between statements within a program.

Statements themselves are written in columns 7 through 72 on both initial and continuation lines.

Lines with a C in column 1 are comment lines. Comment lines do not affect the program but appear in program listings.

Blanks are ignored except for column 6 and certain alphanumeric fields and are used to aid readability.

EXAMPLE: SDS 900 SERIES FORTRAN II PROGRAM

The simple program illustrated in Figure 1 points out many of the properties of an SDS 900 Series FORTRAN II program. It is shown as it would appear on a standard SDS 900 Series FORTRAN II coding form.

The purpose of this program is to solve for the roots of a set of quadratic equations and to output these answers on the typewriter. The ACCEPT TAPE statement reads in N, the number of equations to be solved, and the coefficients (A, B, and C) of each equation. FORMAT No. 1 controls the form of the input. The first record of the input tape contains N and each of the following records contains one set of coefficients A, B, and C. The coefficients are stored in three arrays that are dimensioned 100 each to allow up to 100 equations to be solved. The DO statement allows the portion of the program that solves the equations to be repeated as many times as there are equations to be solved. The IF statement is used to determine if the discriminant of the equation is negative, in which case a portion of the program that calculates complex roots is entered. The TYPE statement types out the answers in the following form:

```
A = 1.0000E 00  B = -3.0000E 00  C = 2.0000E 00
      ROOT1=( 2.0000E 00)+I( 0.          )ROOT2=( 1.0000E 00)+I( 0.          )
A = 1.0000E 00  B =  2.0000E 00  C = 2.0000E 00
      ROOT1=(-1.0000E 00)+I( 1.0000E 00)ROOT2=(-1.0000E 00)+I(-1.0000E 00)
```

After all cases have been completed, the program proceeds out of the DO loop and stops.

BASIC ELEMENTS

QUANTITIES

The SDS 900 Series FORTRAN II compiler is concerned with two types of quantities: integer quantities and floating-point quantities.

Integer quantities are used to represent integers of magnitudes less than 8,388,608.

Floating-point quantities are used to represent the real numbers to a precision of almost 12 decimal digits. The magnitude of a floating-point quantity must be zero or between the limits 10^{77} and 10^{-77}.

CONSTANTS

Integer constants are represented by a string of decimal digits. A maximum of seven digits is allowed, excluding leading zeros.

Floating-point constants are represented by a string of digits which contain a decimal point "." embedded in the string or at either end of the string. A floating-point constant may contain any number of digits; however, only the most significant 12 digits will be used, excluding leading zeros.

EXAMPLES:

1
1973
3.1415926563589

A floating-point constant can be given a scale factor by appending an E followed by an integer constant. The integer constant indicates the power of 10 by which the floating-point constant is to be multiplied and is limited to two digits in length. The magnitude of the resulting number must be between the limits of 10^{-77} and 10^{+77} or be zero.

EXAMPLES:

.1602E-18
299.8E+6
6.02E23

A third alternative allows a floating-point constant to be expressed as an integer constant followed by a scale factor.

EXAMPLES:

5E-2
1E2
1E+76

IDENTIFIERS

Identifiers are used to name the variables, functions, and subroutines which comprise an SDS 900 Series FORTRAN II program. An identifier is a string of letters, or letters and digits, not exceeding seven characters in length. The first character of the string must be a letter.

VARIABLES

Variables may be integer or floating-point, representing respectively integer or floating-point quantities. The identifier used to name an integer variable must begin with I, J, K, L, M, or N. Variables not identified as integer will be considered to be floating-point.

EXAMPLES:

```
SIGMA
BC72
N
```

SUBSCRIPTED VARIABLES

An integer or floating-point variable with subscripts appended is called a subscripted variable and represents a single element in an array of quantities rather than a single quantity. A subscripted variable is denoted by the identifier which names the array, followed by a subscript list enclosed by parentheses. The subscript list is composed of arithmetic expressions separated by commas, each expression corresponding to a subscript. The number of subscripts must equal the dimension of the array.

EXAMPLES:

```
I(3)
X(N)
VOLTAGE(2*N+1,K,K+1)
```

Any expression may be used as a subscript. If the value of a subscript expression is a floating-point number, it will be truncated to an integer before being used as the subscript. The value of a subscript must be greater than zero and not greater than the maximum specified for that array.

EXAMPLES:

```
X(M*N+M-1)
X(THETA)
NUMB(X+MOD(NUMB(X+3),5))
```

LIBRARY FUNCTIONS

The system allows the use of a variety of functions. Here we will consider only the library functions included in the system.

A function acts upon one or more quantities, called its arguments, and produces a single quantity called the function value. A function is denoted by the identifier which names the function, followed by an argument list enclosed in parentheses. The argument list consists of arithmetic expressions separated by commas.

Provision is made for both integer and floating-point functions. Functions producing integer values are integer functions, and functions producing floating-point values are floating-point functions.

Identifiers of integer functions must begin with I, J, K, L, M, or N. Normally, functions not so identified are considered to be floating-point functions. However, to allow programs written for other FORTRAN II systems to be processed, an exception to the standard rule is made. Functions with identifiers of four or more characters, the first of which is "X" and the last "F", are considered to be integer functions.

EXAMPLES:

```
SINF(2*PI*TIME)
ABSF(A)
XMODF(M,K)
```

Many library functions are included in the system as distributed. These include elementary functions such as SINF, EXPF, etc., and arithmetic functions such as ABSF, XMODF, etc.

Library functions are pre-written and constitute "closed" subroutines; that is, they appear only once in the object program, regardless of the number of times they are referenced in the source program.

EXPRESSIONS

FORMATION

An expression is a sequence of constants, variables, and functions separated by operation symbols and parentheses in accordance with mathematical convention and the rules stated below. An expression has a single numerical value, namely, the result of the calculations specified by the arithmetic operations and quantities occurring in the expression.

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

An expression may consist of a single basic element, i.e., a constant, variable, or function. For example:

```
3.1415926
X(N)
SQRTF(ALPHA)
```

Basic elements may be combined through use of the arithmetic operation symbols to form compound expressions, such as:

```
ALPHA+BETA
PI*RADIUS**2
SQRTF(THETA*THETA)
```

Compound expressions may be enclosed in parentheses and regarded as a basic element as follows:

```
(A+B)/(C+D)
((FEET))
POWER(M*(N(K)+1)+1)
```

An entire expression can be preceded by a + or - sign as in:

```
+A
-(-X+Y+Z)
ALPHA(-M,N+1)
```

However, two operation symbols may not appear in sequence. In other words, use the form

```
A*(-B)
```

instead of the illegal form

```
A*-B
```

By repeated use of the above rules, all legal expressions may be constructed.

When the precedence of operations within an expression is not explicitly given by parentheses, it is understood to be the following:

PRECEDENCE	SYMBOL	OPERATIONS
1	**	Exponentiation
2	* and /	Multiplication and Division
3	+ and -	Addition and Subtraction

For example, the expression

```
A**B*C+D
```

is interpreted:

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

Within a precedence group, the omission of parentheses can result in an expression which is considered ambiguous in mathematical notation, e.g., A/B/C. In FORTRAN II, such a sequence is understood to be grouped from the left. For example, if o stands for either * or /, then the expression

```
AoBoCoDoE
```

is interpreted to mean

```
(((AoB)oC)oD)oE
```

The same convention holds for the + and - sign.

EVALUATION

The numerical value of an expression may be of integer or floating-point type. The type of an expression is determined by the types of its constituents. Three cases arise: all constituents are integer (integer expression); all constituents are floating-point (floating-point expression); both types of constituents occur (mixed expression). All of these cases are allowed in the SDS 900 Series FORTRAN II.

INTEGER EXPRESSIONS

An integer expression is evaluated using integer arithmetic throughout, giving an integer value as the result. All results must be limited in magnitude to 8,388,607. Fractional parts arising in division are truncated, not rounded. For example, 5/2 yields 2; 2/3 yields 0.

EXAMPLES:

```
L
I+2*J1
(M+1)*KA-INDEX
```

FLOATING-POINT EXPRESSIONS

Floating-point expressions are evaluated using floating-point arithmetic throughout, yielding a floating-point value.

EXAMPLES:

```
(X(I-1)+X(I+1))/(2.0*DX)
SINF(THETA-ALPHA)
```

MIXED EXPRESSIONS

Mixed expressions are evaluated by first converting all integer quantities to floating-point quantities and then evaluating the expression as if it were a floating-point expression. The result is a floating-point quantity.

EXAMPLES

```
Y+2
Y**N+N*X
A(K)*COSF(2*PI*K/N)
```

PROGRAM STATEMENTS

ARITHMETIC STATEMENTS

The arithmetic statement specifies an expression to be evaluated and a variable (subscripted or non-subscripted) to which the expression value is to be assigned.

FORM: variable = expression

Note that the sign "=" does not mean equality but replacement. The first example below is not an equation but is a valid arithmetic statement meaning "take the value of X, add one, and assign the resulting value to X."

EXAMPLES:

```
X = X+1
Y(I) = SINF(.06*I)
SUM = SUM+TERM*X/N
```

The value of the expression in an arithmetic statement will be made to agree in type with the statement variable before the replacement is performed. Thus, an integer expression value will be converted to a floating-point value if the statement variable is a floating-point variable, and a floating-point expression value will be truncated to an integer if the statement variable is an integer variable.

CONTROL STATEMENTS

The normal flow of a FORTRAN II program is sequentially through the statements in the order in which they are presented to the compiler. Control statements allow the programmer to specify the flow of the program. To this end, statements can be given numbers to be referenced by control statements. These statement numbers must be unique.

Unconditional GO TO Statement

FORM: GO TO n

where n is a statement number.

This statement transfers control to the statement numbered n.

EXAMPLE:

```
GO TO 15
```

Computed GO TO Statement

FORM: GO TO $(n_1, n_2, n_3, \ldots, n_k)$, expression

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

This statement transfers control to statement $n_1, n_2, \ldots, n_k$ depending on whether the expression has the value $1, 2, \ldots, k$, respectively. The value of the expression will be converted to an integer if required. The statement is not defined for expression values other than $1, 2, \ldots, k$.

EXAMPLES:

```
GO TO (1,2,3,4), K
GO TO (13,27,1,4,6), V(J)
```

IF Statement

FORM: IF (expression) n_1, n_2, n_3

where n_1, n_2, n_3 are statement numbers.

This statement transfers control to the statement n_1, n_2, or n_3 if the value of the expression is, respectively, less than, equal to, or greater than, zero.

EXAMPLES:

```
IF (A(I)-ARG) 5,2,4
IF (Y) 14,15,15
```

DO Statement

FORMS: DO n variable = $expression_1$, $expression_2$

DO n variable = $expression_1$, $expression_2$, $expression_3$

where n is a statement number.

If $expression_3$ is not stated (first form), it is understood to be 1.

This statement causes the statements that follow, up to and including statement n, to be executed repeatedly. This group of statements is called the range of the DO statement. Initially, the statements of the range are executed with the value of $expression_1$ assigned to the variable. This initial execution is always performed, regardless of the values of $expression_2$ and $expression_3$. After each execution of the range, the value of $expression_3$ is added to the value of the variable and the result is compared with the value of $expression_2$. If the value of the variable is not greater than the value of $expression_2$, the range is executed again using the new value of the variable. In case the value of $expression_3$ is negative, another execution will be performed if the new value of the variable is not less than the value of $expression_2$. After the last execution, control passes to the statement immediately following statement n. This exit from the range of the DO statement is called the normal exit. Exit may also be effected by a transfer from within the range of the DO statement.

The variable of the DO statement is available for use throughout the range of the statement and, if a transfer exit occurs, the variable retains its current value for subsequent use. The value of the DO variable is not defined, however, for a normal exit. The range of a DO statement may include other DO statements provided that the range of each "inside" DO statement is contained completely within the range of an "outside" DO statement. In other words, the ranges of two DO statements may not partially intersect one another. Only total intersection or no intersection is allowed.

No transfer into the range of a DO statement from outside of its range is permitted.

EXAMPLES:

```
DO 2  L = 1,N
DO 5  V = POINT1,END,DY
DO 16 N(K+1) = O,K
```

CONTINUE Statement

FORM: CONTINUE

This statement is a dummy, or "do nothing", statement used primarily to serve as a target point for transfers, particularly as the last statement in the range of a DO statement. For example, in the statement sequence:

```
      DO 5   I = 1,MAX
             .
             .
             .
      GO TO 5
             .
             .
             .
      X = SUM
             .
             .
             .
5     CONTINUE
             .
             .
             .
```

if the GO TO is intended to begin another execution of the DO range, without performing the statement X = SUM, the CONTINUE statement provides the necessary target address.

SPECIFICATION STATEMENTS

Specification statements are used to supply information used for storage allocation. This information may be required by the system, or merely supplemental. Supplemental information is used to reduce the storage requirements of the program.

DIMENSION Statement

FORM: DIMENSION variable, variable, ..., variable

Each variable of a DIMENSION statement is subscripted with one or more integer constants which specify the maximum value the corresponding subscript may assume. For example, the statement:

```
DIMENSION  X(10),Y(10,20)
```

specifies a maximum of 10 for a subscript of the variable X and a maximum of 10 and 20 respectively for the first and second subscripts of the variable Y.

The information provided by a DIMENSION statement is required for allocation of storage for arrays. Each subscripted variable which appears in a program must also appear in a DIMENSION statement. The DIMENSION statement for a variable must precede the first appearance of that variable in the program.

EXAMPLES:

```
DIMENSION  ALPHA(5)
DIMENSION  A(10,10,10),B(10,20,30)
```

EQUIVALENCE Statement

FORM: EQUIVALENCE (identifier, identifier, ...), (identifier, identifier, ...), ...

The identifiers of an EQUIVALENCE statement may be simply the names of variables or arrays, or identifiers appended by a single parenthesis-enclosed integer constant. The inclusion of two or more identifiers in a parenthesis pair specifies that these quantities share the same storage location.

EXAMPLE:

```
EQUIVALENCE  (HOGAN,GOAT)
```

This statement specifies that quantities HOGAN and GOAT are to share the same storage location.

When program logic permits, the number of storage locations required by the program can be reduced through use of the EQUIVALENCE statement.

To identify a specific location in an array, that location may be appended as an integer constant to the array identifier. For example, if ALPHA is a variable and BETA is an array, the statement

EQUIVALENCE (ALPHA,BETA(4))

specifies that ALPHA and the fourth location of array BETA are to share the same storage location.

To identify a specific quantity in a multiply-dimensioned array, the location of that quantity must first be calculated. For example, consider a three-dimensional array specified by

DIMENSION CUBE(n_1,n_2,n_3)

where n_1, n_2, and n_3 are the maximum subscript values permitted with this array. To calculate the location of the quantity

CUBE(k_1,k_2,k_3)

use the formula

location = $(k_3-1)*n_1*n_2+(k_2-1)*n_1+k_1$

thus, the statement pair

DIMENSION TEMP(10),CUBE(2,4,12)
EQUIVALENCE (TEMP(4),CUBE(15))

specifies that the quantities TEMP(4) and CUBE(1,4,2) are to share the same storage location.

When the location of a variable is known relative to a second variable, this location may be specified by appending an integer constant to the identifier of the second variable. The integer to be used can be determined by considering a sequence of quantities as a one-dimensional array. For example, if we have in storage at

LOCATION L_1: ALPHA
L_2: BETA
L_3: GAMMA
L_4: DELTA

then the statement

EQUIVALENCE (X,ALPHA(3))

specifies that the quantity X and GAMMA are to share the same storage location.

Note, this property of equivalence is transitive; in other words, both of the statements

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

specify that A, B, and C are to share the same storage location.

COMMON Statement

FORM: COMMON identifier,identifier,....,identifier

The identifiers of a COMMON statement are names of variables or arrays. The COMMON statement specifies that the variables and arrays indicated are to be stored in an area also available to other programs. By use of COMMON statements, a common storage area may be shared by a program and its subprograms.

Each array name which appears in a COMMON statement must also appear in a DIMENSION statement in the same program.

Quantities whose identifiers appear in COMMON statements will be allocated storage in the same sequence that their identifiers appear in the COMMON statements, beginning with the first COMMON statement in the program.

Storage allocation for common quantities begins at the same location for all programs. Thus, the programmer can establish a one-to-one correspondence between the quantities of several programs even when the same quantities have different identifiers in the different programs. For example, if a program contains

```
COMMON  A,B,C
```

as its first COMMON statement, and a subprogram has

```
COMMON  X,Y,Z
```

as its first COMMON statement, then A and X will refer to the same storage location. A similar correspondence holds for the pairs B and Y, C and Z.

An exception to this order of allocation occurs when a variable or array name appears in both a COMMON and an EQUIVALENCE statement. In this case, priority is given to those appearing in EQUIVALENCE statements in the order in which they appear.

EXAMPLES:

```
COMMON  A,B,C,X,Y,Z
COMMON  ALPHA,THETA,MATRIX
```

INPUT-OUTPUT STATEMENTS

Input-output statements call for the transmission of information between computer storage and various input-output units such as the console typewriter, magnetic tapes, paper tapes, and so on. The first requirement is to name the operation required, such as READ INPUT TAPE 3 or TYPE. Next, a data format must be specified by a FORMAT statement number and, finally, a list of variables whose values are being transmitted is specified. The listed order of the variables must be the same as the order in which the information exists on the input medium or will exist on the output medium. Consider the statement:

```
TYPE 6, ALPHA,BETA,GAMMA
```

The statement says, "type on the console typewriter the values of the variables ALPHA, BETA, and GAMMA in that order and according to the format specified by the FORMAT statement numbered 6."

Likewise, the statement

```
ACCEPT 4, WINE,WOMEN,SONG
```

says, "accept from the console typewriter the values of WINE, WOMEN, and SONG according to format 4."

Indexing within input-output lists follows rules similar to those used with DO statements. For example,

```
TYPE 8, (FORCE(I),I=1,3)
```

is equivalent with

```
TYPE 8, FORCE(1),FORCE(2),FORCE(3)
```

Indexing of this nature can be compounded as in the following example:

```
ACCEPT TAPE 2, ((TRIX(I,J),I=1,M)J=1,N)
```

This statement accepts from paper tape an M by N matrix in the order

```
TRIX(1,1),TRIX(2,1),...,TRIX(M,1),TRIX(1,2),...,TRIX(M,N)
```

Indexing identifiers used in this way are dummy identifiers and will not affect variables so identified elsewhere in the program.

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

```
ACCEPT TAPE  2, TRIX
```

On input lists of the form K,A,(B(K)) or K,(A),B(K) where an indexing variable appears early in the list, indexing will be carried out with the new value only if:

1) the subscripted variable is enclosed in parenthesis, or;
2) a parenthesis-enclosed variable, or a list of variables enclosed in parenthesis, separates the indexing variable and the subscripted variable.

ACCEPT Statement

FORM: ACCEPT n, list

This statement causes information to be read from the console typewriter and put into storage as values of the variables in the list. The data is converted from external to internal form as specified by FORMAT statement n.

EXAMPLE:

```
ACCEPT 14, A,I
```

TYPE Statement

FORM: TYPE n, list

This statement causes the values of variables in the list to be read from storage and typed on the console typewriter. The data is converted from internal to external form as specified by FORMAT statement n.

EXAMPLE:

```
TYPE 14, A,B,C
```

PRINT Statement

FORM: PRINT n, list

This statement causes the values of variables in the list to be read from storage and printed on the on-line printer. The data is converted from internal to external form as specified by FORMAT statement n.

EXAMPLE:

```
PRINT 3, A,B,C
```

ACCEPT TAPE Statement

FORM: ACCEPT TAPE n, list

This statement causes information to be read from paper tape and put into storage as values of the variable in the list. The data is converted from external to internal form as specified by FORMAT statement n.

EXAMPLE:

```
ACCEPT TAPE 17, A,(B(J),J=1,11)
```

PUNCH TAPE Statement

FORM: PUNCH TAPE n, list

This statement causes the values of variables in the list to be read from storage and punched on paper tape. The data is converted from internal to external form as specified by FORMAT statement n.

EXAMPLE:

```
PUNCH TAPE 2, A,I,B(2,1)
```

READ INPUT TAPE Statement

FORM: READ INPUT TAPE expression, n, list

This statement causes BCD information to be read from a magnetic tape unit and put in storage as values of the variables in the list. The number of the tape unit will be equal to the value of the expression, truncated if necessary. The data is converted from external to internal form as specified by FORMAT statement n.

EXAMPLES:

```
READ INPUT TAPE  3, 2, A,B,C
READ INPUT TAPE  A(K)+2, 5, A,I,B
```

WRITE OUTPUT TAPE Statement

FORM: WRITE OUTPUT TAPE expression, n, list

This statement causes the values of variables in the list to be read from storage and written on magnetic tape in BCD form. The number of the tape unit will be equal to the value of the expression, truncated if necessary. The data is converted from internal to external form as specified by FORMAT statement n.

EXAMPLES:

```
WRITE OUTPUT TAPE  3, 5, A
WRITE OUTPUT TAPE  K, 5, (A(I),B,I=1,10)
```

READ Statement

FORM: READ n, list

This statement causes information to be read from punched cards and put in storage as values of the variables in the list. The data is converted from external to internal form as specified by FORMAT statement n.

EXAMPLE:

```
READ  121, A, I,B
```

PUNCH Statement

FORM: PUNCH n, list

This statement causes the values of variables in the list to be taken from storage and punched on cards. The data is converted from internal to external form as specified by FORMAT statement n.

EXAMPLE:

```
PUNCH  123, ((A(I,J),I=1,10),J=2,14,2)
```

READ DRUM Statement

FORM: READ DRUM $expression_1$, $expression_2$, list

This statement causes binary information to be read from a drum and put in storage as values of the variables in the list. The number of the drum will be equal to the value of $expression_1$, truncated if necessary. The drum address of the first word transmitted from the drum will be equal to the value of $expression_2$, truncated if necessary.

EXAMPLES:

```
READ DRUM  2, 450, A,B,C
READ DRUM  K+1, B*C, A,G(I)
```

WRITE DRUM Statement

FORM: WRITE DRUM $expression_1$, $expression_2$, list

This statement causes the values of variables in the list to be read from storage and written on a drum in binary form. The number of the drum will be equal to the value of expression$_1$, truncated if necessary. The drum address of the first word transmitted to the drum will be equal to the value of expression$_2$, truncated if necessary.

EXAMPLES:

```
WRITE DRUM  1, 75, (A(I),I=1,50)
WRITE DRUM  K, L, A,B,C
```

READ TAPE Statement

FORM: READ TAPE expression, list

This statement causes binary information to be read from a magnetic tape unit and put in storage as values of the variables in the list. The number of the tape unit will be equal to the value of the expression, truncated if necessary.

EXAMPLES:

```
READ TAPE  3, A,B
READ TAPE  K, A,B,C
```

WRITE TAPE Statement

FORM: WRITE TAPE expression, list

This statement causes the values of variables in the list to be read from storage and written on magnetic tape in binary form. The number of the tape unit will be equal to the value of the expression, truncated if necessary.

EXAMPLES:

```
WRITE TAPE  3, A,B
WRITE TAPE  K+3, A,B,C
```

BACKSPACE Statement

FORM: BACKSPACE expression

This statement directs a magnetic tape unit to backspace a record. The number of the tape unit will be equal to the value of the expression, truncated if necessary.

EXAMPLES:

```
BACKSPACE  3
BACKSPACE  K+1
```

REWIND Statement

FORM: REWIND expression

This statement directs a magnetic tape unit to rewind the tape. The number of the tape unit will be equal to the value of the expression, truncated if necessary.

EXAMPLES:

```
REWIND  3
REWIND  ALPHA
```

END FILE Statement

FORM: END FILE expression

This statement directs a tape unit to write an end-of-file mark on the tape. The number of the tape unit will be equal to the value of the expression, truncated if necessary.

EXAMPLE:

```
END FILE  3
```

FORMAT STATEMENTS

All input or output activity requires the use of a FORMAT statement to specify the format of the data and the type of conversion to be used.

FORM: FORMAT $(s_1, s_2, \ldots, s_k)$

where s is a data field specification.

For sake of clarity, examples given below refer to the console typewriter. However, any FORMAT statement can be used with any input-output medium (magnetic tape, paper tape, punched cards, and console typewriter).

Numerical Fields

Conversions of numerical data during input-output may be one of three types:

1) type-E
   internal form - binary floating-point
   external form - decimal floating-point

2) type-F
   internal form - binary floating-point
   external form - decimal fixed-point

3) type-I
   internal form - binary integer
   external form - decimal integer

These types of conversions are specified by the forms:

1) Ew.d
2) Fw.d
3) Iw

where E, F, and I specify the type of conversion required, w is an integer specifying the width of the field, and d is an integer specifying the number of decimal places to the right of the decimal point.

As an example, in using the statement

FORMAT (I8,F8.3,E15.6)

the line

32 4.263 -0.186214E-22

might be typed on the console typewriter.

Scale Factors

Scale factors can be specified for F and E type conversions. A scale factor has the form nP where P is the control or identifying character, and n is a signed or unsigned integer specifying the scale factor. In F type conversions, the scale factor specifies a power of ten, such that

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

With E type conversions, the scale factor is used to change the number by a power of ten and then to correct the exponent such that the result represents the same real number as before, but now has a different form. For example, if the statement

FORMAT (F10.3,E14.4)

corresponds to the line

14.614 -0.6861E-00

then the statement

FORMAT (-2PF10.5,1PE14.3)

will correspond to the line

.14614 -6.861E-01

The scale factor is assumed zero if none has been given. However, once a value has been given, it will hold for all E and F type conversions following the scale factor. A zero scale factor can be used to return conditions to normal. Scale factors have no effect on type I conversions.

Alphanumeric Fields

Alphanumeric data can be handled in much the same manner as numeric data through use of the form Aw where A is the control character and w is the number of characters in the field. Consider:

FORMAT (A5)

During input, this statement is used to accept five characters from the input record. During output, five characters would be included in the output record.

Alphanumeric Format Fields

Alphanumeric fields may be specified within a FORMAT statement by use of the form kH followed by k alphanumeric characters, counting blanks. During input, k characters are extracted from the input record and replace the k characters included within the specifications. During output, the k characters specified, or the k characters which have replaced them, become part of the output record.

For example, the statement

FORMAT (15H TEST COMPLETE)

can be used to type

TEST COMPLETE

on the console typewriter. Note, the FORMAT statement above can be used to replace the TEST COMPLETE comment with a new comment from an input record and then used again to type this new comment on the console typewriter.

Mixed Fields

An alphanumeric format field specification may be followed by any field specification to form a mixed field specification. For example, the use of the statement

FORMAT (12H VELOCITY = F8.4)

can result in the output line

VELOCITY = 6.4142

An alphanumeric format field specification can also be followed by the repeated field and multiple record specifications outlined below.

Blank or Skip Fields

The specification kX may be used to include k blank characters in an output record, or to skip k characters of an input record. Consider:

FORMAT (4HTIMEF8.4, 12X, 1HXF8.2)

This statement can be used to output

TIME 1.2863 X -148.61

where twelve blanks separate the two quantities.

Repetitions of a Field Specification

It may be desired to input or output successive fields within one input or output record according to the same field specifications. This is done by preceding the control character (E, F, I, or A) by the number of repetitions (k) desired. Thus, the statement

FORMAT (12A6)

specifies during input that twelve fields of six characters each are to be accepted from the input record.

Repetition of Groups

Parentheses can be used for repetition of groups of field specifications. Thus, the statement

FORMAT (2(E6.1,F10.6),F6.6)

is equivalent with

FORMAT (E6.1,F10.6,E6.1,F10.6,F6.6)

Multiple Record Specifications

To handle a file of input-output records (a page of printed lines, a deck of cards, etc.) where different records have different field specifications, a virgule "/" is used to indicate a new record. Thus, the statement

FORMAT (2F6.4/13,F6.4)

is equivalent to the statement

FORMAT (2F6.4)

for record one, and the statement

FORMAT (I3,F6.4)

for record two.

If the field specifications of the first record are different from that of following records (master record at the start of a file, etc.), then the field specifications of the first record (master record) should be followed by the field specifications of the following records (data records) enclosed in parentheses as shown in the statement below.

FORMAT (6I10,F12.2/(6E12.0))

In general, if transmission of data is to continue (as specified by the variable list of an input-output statement) when the end of a format statement (except for parentheses) has been reached, the format is repeated on the next input-output record from the last open parenthesis. Thus, both the virgule and the sequence of closing parenthesis at the end of a FORMAT statement indicate the termination of a record.

Blank lines may be introduced in printed text by using consecutive virgules.

ARITHMETIC FUNCTION DEFINITION STATEMENTS

FORM: identifier(identifier,identifier,....) = expression

This statement serves to define a function for use in a particular program, and the function definition holds only in the program containing the definition. The appearance of the function name in an expression suffices to call the function, as in the case of library functions. The function will have a single value whose type will be determined by the function identifier.

The defining expression for a function may include other previously defined functions or library functions.

The list of identifiers enclosed in parenthesis represents the argument list of the function. These identifiers must agree in number, order, and type with the actual arguments which will be present when the function is used. An argument of the function is specified in the defining expression through use of its corresponding identifier.

Identifiers of arguments are dummy identifiers. They have meaning and must be unique only within the definition statement and may be identical to identifiers appearing elsewhere in the program.

Identifiers which represent quantities other than arguments of the function can be used in the defining expression. These quantities will act as parameters, i.e., the function will be evaluated using values which are current at the time the function is called.

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

EXAMPLES:

```
NDAYS(I) = 7*(I/5)+XMODF(I,5)
AV(X,Y) = (X+Y)/2.
DER(X,I) = AV(X(I-1),X(I+1))/DELTAT
```

SUBPROGRAM STATEMENTS

A program written in FORTRAN II language that is referred to or called by another FORTRAN II program is called a subprogram. Subprograms are complete programs, conforming to all rules of FORTRAN II programming. They may be compiled independently or with the main program which refers to them. A subprogram can call other subprograms during its execution, however, recursion is not permitted.

Two types of subprograms are available: the FUNCTION subprogram and the SUBROUTINE subprogram. The use of the statements FUNCTION, SUBROUTINE, RETURN, and CALL in the definition and use of subprograms is described below.

FUNCTIONS

A FUNCTION subprogram, like a library or an arithmetic function, is single-valued and is called or referred to by the appearance of its name in an expression. A FUNCTION subprogram begins with a FUNCTION statement and returns control to the main program by means of one or more RETURN statements.

```
FORM:   FUNCTION   identifier(identifier,identifier,...)
            .
            .
            .
        RETURN
            .
            .
            .
        RETURN
            .
            .
            .
        END
```

FUNCTION Statement

```
FORM:   FUNCTION   identifier(identifier,identifier,...)
```

This statement must be the first statement of a FUNCTION subprogram. The first identifier is the name of the function being defined. Identifiers appearing on the list enclosed in parenthesis are dummy identifiers which represent the arguments of the function. These identifiers must agree in number, order, and type with the actual arguments which will be presented to the function when it is called. For example, when a dummy identifier represents an integer array name, the corresponding actual argument must be an integer array name.

Dummy identifiers which represent the names of arrays must appear in DIMENSION statements in the subprogram. Furthermore, the declared dimension of each must equal the dimension of the actual arrays specified when the function is called.

None of these dummy identifiers may appear in a COMMON or EQUIVALENCE statement in the subprogram.

A function must have at least one argument. The value of the function returned to the calling program is the value assigned to the function identifier during execution of the function.

EXAMPLES:

```
FUNCTION        FIND(TABLE, X)
FUNCTION        MEMBER(SET, FORM)
```

SUBROUTINES

A SUBROUTINE subprogram may be multi-valued and can only be referred to by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE statement and returns control to the main program by means of one or more RETURN statements.

```
FORM:    SUBROUTINE   identifier(identifier, identifier, ...)
                .
                .
                .
         RETURN
                .
                .
                .
         RETURN
                .
                .
                .
         END
```

SUBROUTINE Statement

```
FORM:    SUBROUTINE   identifier(identifier, identifier, ...)
```

The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram. The first identifier is the name of the subroutine. The identifiers appearing on the list enclosed in parenthesis are dummy identifiers which represent the arguments of the subroutine. These identifiers must agree in number, order, and type with the actual arguments which will be presented to the subroutine when it is called, e.g., if a dummy identifier is used as a floating-point array name, the corresponding argument must also be a floating-point array name.

Dummy identifiers which represent array names must appear in DIMENSION statements in the subprogram. The dimensions so declared must equal the corresponding dimension of the actual arrays specified when the subroutine is called.

None of the dummy identifiers of a SUBROUTINE statement may appear in a COMMON or EQUIVALENCE statement within the subprogram.

A SUBROUTINE subprogram may use any of its dummy identifiers to represent results or values of the subroutine.

EXAMPLES:

```
SUBROUTINE      FACTOR(COEF1, COEF2, COEF3, ROOT1, ROOT2)
SUBROUTINE      DOT(N, V1, V2, V3)
```

CALL Statement

```
FORM:    CALL   identifier(expression, expression, ...)
```

This statement is used to call, or transfer control to, a SUBROUTINE subprogram. The identifier is the name of the subroutine and the expressions specify the arguments the subroutine is to use. Array names used as arguments must refer to arrays of dimension equal to the dimension declared for the corresponding dummy identifier in the subprogram.

EXAMPLES:

```
CALL        FACTOR(A+1, 2*COSF(THETA)*B(I), C(I), R1, R2)
CALL        DOT(2*MARK, X, Y, VNORM)
```

RETURN Statement

FORM: RETURN

This statement returns control from the subprogram to the calling program. Thus, the last statement executed in a subprogram will be a RETURN statement. It need not be physically the last statement in a program, but can be at any point in the subprogram at which it is desired to terminate execution. Any number of RETURN statements can be used.

ADDITIONAL STATEMENTS

Assigned GO TO Statement

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

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

This statement transfers control to the statement whose number is equal to the current value of the variable. The current value of the variable is determined by the last executed ASSIGN statement in which the variable appears and must be one of the integers $n_1, n_2, \ldots, n_k$. The variable must appear in a previously executed ASSIGN statement.

EXAMPLES:

```
GO TO  L, (1,3,10)
GO TO  ENTRY, (5,4,14,23)
```

ASSIGN Statement

FORM: ASSIGN integer TO variable

This statement sets the value of the variable for a subsequent assigned GO TO statement. The integer must be one of the statement numbers allowed by the assigned GO TO statement in which the variable appears.

EXAMPLES:

```
ASSIGN  2 TO K
ASSIGN  14 TO ENTRY
```

SENSE LIGHT Statement

FORM: SENSE LIGHT expression

During compilation, a storage cell, initialized to zero, is set aside for flags. This statement causes one bit of this cell to be set to one. The particular bit chosen is specified by the value of the expression, truncated if necessary. The integer so derived must be one of the integers 1, 2, ..., 24.

EXAMPLES:

```
SENSE LIGHT  3
SENSE LIGHT  2*X+1
```

IF SENSE LIGHT Statement

FORM: IF(SENSE LIGHT expression) n_1, n_2

where n_1, n_2 are statement numbers.

This statement transfers control to statement n_1 or n_2 depending on whether a bit in the flag cell is one or zero. The particular bit tested is specified by the value of the expression, truncated if necessary. The resulting integer must be one of the integers 1, 2, ..., 24.

EXAMPLES:

```
IF(SENSE LIGHT  3)    1,2
IF(SENSE LIGHT  2*K/3)  12,7
```

IF SENSE SWITCH Statement

FORM: IF(SENSE SWITCH expression) n_1,n_2

where n_1 and n_2 are statement numbers.

This statement transfers control to statement n_1 or n_2 depending on whether a sense switch is ON or OFF. The particular sense switch used is specified by the value of the expression, truncated if necessary. The resulting integer must be 1, 2, 3, or 4.

EXAMPLE:

```
IF(SENSE SWITCH  3)    1,2
IF(SENSE SWITCH  K+2)  14,5
```

IF ACCUMULATOR OVERFLOW Statement

FORM: IF ACCUMULATOR OVERFLOW n

where n is a statement number.

This statement tests the toggle which indicates accumulator overflow on addition. If the toggle indicates that an overflow has occurred, control is transferred to statement n.

EXAMPLE:

```
IF ACCUMULATOR OVERFLOW  10
```

IF DIVIDE CHECK Statement

FORM: IF DIVIDE CHECK n

where n is a statement number.

This statement tests the toggle which indicates a division by zero. If the toggle indicates that a zero division has occurred, control is transferred to statement n.

EXAMPLE:

```
IF DIVIDE CHECK  5
```

PAUSE Statement

FORMS: PAUSE
PAUSE expression

This statement halts the machine. Program execution may be resumed by depressing the START key. The value of the expression will be displayed on the console.

EXAMPLES:

```
PAUSE
PAUSE  K
```

STOP Statement

FORMS: STOP
STOP expression

This statement halts the machine and prints the word "STOP" on the console typewriter. Depression of the START key will have no effect. The value of the expression will be displayed on the console.

EXAMPLES:

```
STOP
STOP Q
```

COMPATABILITY STATEMENTS

FREQUENCY Statement

The FREQUENCY statement is used in some other FORTRAN II systems to provide information for efficient index register assignment. It is not required by the SDS 900 Series FORTRAN II system but is allowed to insure acceptability of programs written for these systems.

IF QUOTIENT OVERFLOW Statement

The IF QUOTIENT OVERFLOW statement finds use in certain FORTRAN II systems to offset a peculiar mechanization of floating-point arithmetic. This statement has no use in the SDS 900 Series FORTRAN II processor but is allowed to insure acceptability of programs written for these systems. The IF QUOTIENT OVERFLOW statement is executed at run time as if quotient overflow never occurs.

SDS 900 SERIES FORTRAN II STATEMENTS

1. Arithmetic Statement
2. Arithmetic Function Definition Statement
3. ACCEPT Statement
4. ACCEPT TAPE Statement
5. ASSIGN Statement
6. Assigned GO TO Statement
7. BACKSPACE Statement
8. CALL Statement
9. COMMON Statement
10. Computed GO TO Statement
11. CONTINUE Statement
12. DIMENSION Statement
13. DO Statement
14. END FILE Statement
15. EQUIVALENCE Statement
16. FORMAT Statement
17. FREQUENCY Statement
18. FUNCTION Statement
19. GO TO Statement
20. IF ACCUMULATOR OVERFLOW Statement
21. IF DIVIDE CHECK Statement
22. IF QUOTIENT OVERFLOW Statement
23. IF Statement
24. IF SENSE LIGHT Statement
25. IF SENSE SWITCH Statement
26. PAUSE Statement
27. PRINT Statement
28. PUNCH Statement
29. PUNCH TAPE Statement
30. READ Statement
31. READ DRUM Statement
32. READ INPUT TAPE Statement
33. READ TAPE Statement
34. RETURN Statement
35. REWIND Statement
36. SENSE LIGHT Statement
37. STOP Statement
38. SUBROUTINE Statement
39. TYPE Statement
40. WRITE DRUM Statement
41. WRITE OUTPUT TAPE Statement
42. WRITE TAPE Statement

SDS
SCIENTIFIC DATA SYSTEMS

1649 Seventeenth Street □ Santa Monica, California □ UPton 0-5471