XDS

Xerox Data Systems

Reference Manual

XDS FORTRAN II

90 00 03D

Price: $3.00

FORTRAN II
REFERENCE MANUAL

for

XDS 900 SERIES COMPUTERS

90 00 03D

October 1970

XDS

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

Printed in U.S.A.

REVISION

This publication, 90 00 03D, is a minor revision of the XDS 900 Series FORTRAN II Reference Manual, 90 00 03C, dated February 1967. Changes to the previous edition are indicated by a line in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
XDS 910 FORTRAN II Operations Manual	90 00 11
XDS 920/930 FORTRAN II Operations Manual	90 00 46

NOTICE

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of sume features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their XDS sales representative for details.

CONTENTS

FIGURES

INTRODUCTION

This manual is intended as a reference manual for the XDS 900 Series FORTRAN II System and assumes the reader is familiar with the general principles of FORTRAN programming.

The XDS 900 Series FORTRAN II language provides engineers and scientists with an efficient and easily understood means of writing programs for the XDS 900 Series computers. Programming is accomplished by the use of expressions which resemble accepted mathematical notations, allowing the programmer to concentrate on the problem to be solved rather than the details of computer operation. In addition, features are included for use at run time and compile time to reduce the cost and time required for program checkout.

The XDS 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 are a direct subset of the XDS 900 Series FORTRAN II processor.

With the provision that reasonable restrictions are met, the XDS 900 Series FORTRAN II processor will compile and run FORTRAN II programs written for other computers.

These restrictions are:

1. The memory capacity of the XDS 900 Series computer must be sufficient to hold the compiled program and all subroutines required at run time. Normally, less memory will be required on the XDS 900 Series System than on other systems.
2. All peripheral equipment (such as magnetic tapes) called for in the program, must be attached to the XDS 900 Series computer. The system checks for the presence of required equipment.
3. Integer quantities are limited to 8,388,607 and floating-point precision is limited to approximately twelve decimal digits. In general, these precisions exceed those of other systems.

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 given in this manual. Most illegal programs will be caught by the system.

Only a basic XDS 900 Series Computer, with 4096 words of core memory, paper tape and typewriter input/output, is required for complete processing and solution of FORTRAN II programs.

For 4096-word configurations, the only important limitation on source program size is the number of distinct symbols and labels used in the program. Programs with as many as 325 symbols and labels may be compiled on the XDS 920/930 and programs with as many as 200 on the XDS 910/925. In practice, this is no limitation since the number of labels may be expected to be proportional to the program size, and no practical limitation exists on the size of FORTRAN programs which may be compiled on an XDS 900 Series Computer with greater than 4096 words of core memory.

Refer to the appropriate FORTRAN II Operations Manual for the operating description of this system:

Manual	XDS Publication No.
XDS 910 FORTRAN II Operations	900011C
XDS 920/930 FORTRAN II Operations	900046D

I. XDS 900 SERIES FORTRAN II PROGRAMS

An XDS 900 Series FORTRAN II program consists of a sequence of statements which specify the procedure to be followed by the computer. These statements fall into four general categories:

INPUT/OUTPUT statements which call for transmission of information between computer storage and various input-output devices.
ARITHMETIC statements which indicate calculations to be performed.
CONTROL statements that determine the sequence in which statements will be performed.
DECLARATION statements that supply information about the program rather than specifying operations.

PROGRAM PREPARATION

The sequence of statements comprising an XDS 900 Series FORTRAN II program is written on a coding form. This information is then punched on cards or paper tape for entry into the computer. The same coding form is used for either input medium.

Figure 1 illustrates an XDS 900 Series FORTRAN II program written on a standard XDS 900 Series FORTRAN II coding form. Each statement of the program is written on a separate line; however, a statement too long to fit on one line may utilize as many as three continuation lines.

Each line of the coding form is divided into 72 spaces or columns and each space may contain one character. When cards are used as the input medium, each line of the coding form corresponds to a card and each space to a card column. Figure 2 illustrates the statement on line 6 of the example as it would appear on a punched card.

The columns of the coding form are grouped into fields. The first field, columns 1 through 5, is used for the statement number, if any. These numbers permit cross reference between statements within a program. Blanks and leading zeros in this field are ignored.

PROBLEM _____

XDS
Xerox Data Systems

_____

FORTRAN CODING SHEET

PROGRAMMER _____

73 Identification 80

PAGE _____ OF _____

DATE _____

C FOR COMMENT

STATEMENT NUMBER 1–5	Cont. 6	FORTRAN STATEMENT 7–72
*	EX	AMPLE 1
		DIMENSIθN PRICES(100)
		READ 5, PRICES
		SUM = 0
		Dθ 2 K=1,100
2		SUM = SUM + PRICES(K)
		AVERAGE = SUM/100.0
		TYPE 5, AVERAGE
		STθP
5		FθRMAT(F10.2)
		END

Figure 1. FORTRAN II Sample Program

SDS-B-144

The second field, columns 7 through 72, is used for the statement itself. Except for certain alphanumeric strings, blanks in this field are ignored and are used to aid readability.

Column 1 serves another function, that of specifying comment lines and compiler control lines. A "C" in column 1 indicates that the line is a comment and is not to be processed. Comments appear in program listings but do not otherwise affect the program. An asterisk, "*", in column 1 indicates that the line is a compiler control line. No control lines are necessary, but any number of them may precede the program. They have no effect, but will be listed even when program listing is suppressed.

EXAMPLE

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

The purpose of the program is to find the average of a set of 100 prices and to type the average on the console typewriter.

Line 1 is a control line.

The DIMENSION statement of line 2 declares PRICES to be an array of 100 numbers. This declaration causes storage sufficient for 100 numbers to be set aside and allows PRICES to appear with subscripts in the program.

Line 3 is an input statement which reads numbers from punched cards and places them in consecutive locations of the array PRICES. Reading will continue until all of the 100 values declared for PRICES have been entered.

The READ statement refers to FORMAT statement number 5 (line 10) for specification of how prices are punched on the cards. The FORMAT specifies that the prices are punched one to a card, occupy ten columns each, and have two digits to the right of the decimal point. A typical input card, pictured in Figure 3, gives $1.85 as the price of May wheat.

The calculation proper begins with the assignment statement of line 4. This statement sets

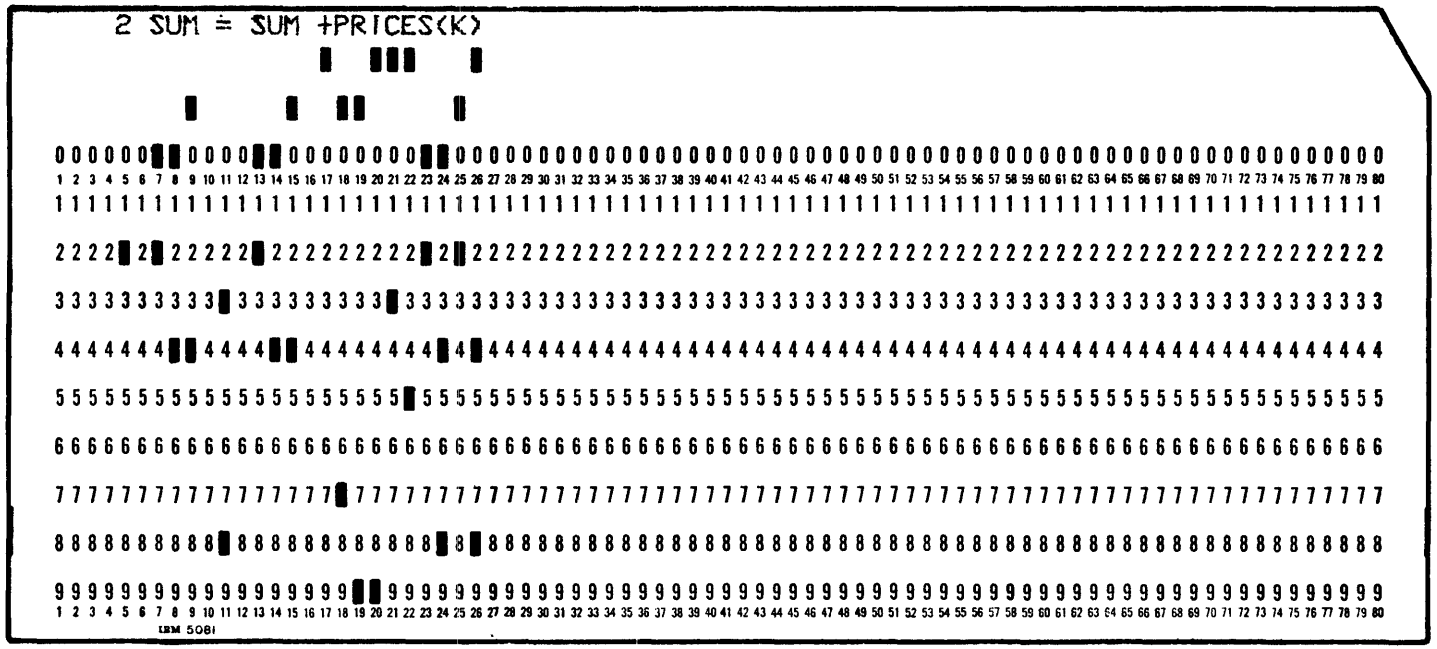

Figure 2. Example of FORTRAN Statement

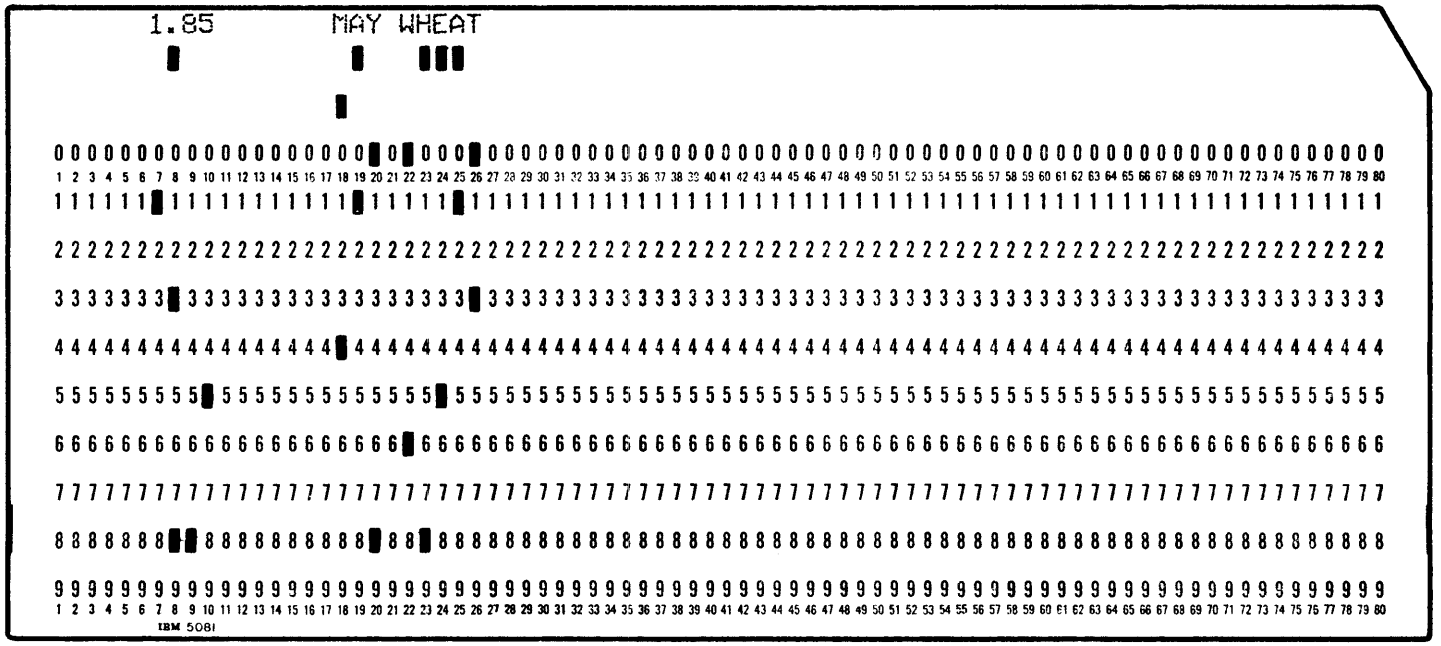

Figure 3. Typical Input Card

the variable SUM to zero as an initial value.

Line 5 is a control statement (called a DO statement) which causes the statement following (number 2) to be executed 100 times. The variable K is set to one for the first execution and then is increased by one for each subsequent execution. The last time statement 2 is performed, K has the value 100.

Statement 2 adds one of the PRICES to SUM and assigns the result to SUM. The particular price used from the array of PRICES will be determined by the value of K, which appears as a subscript. The first execution uses PRICES (1), the first number in the array; next time PRICES (2) is added in, and so on. After PRICES (100) has been added to SUM, the program proceeds to the next statement.

At this point SUM contains the sum of the numbers read in from cards. The assignment statement of line 7 now divides this sum by 100 and assigns the result to AVERAGE. This, of course, is the answer.

The output statement of line 8 causes the value of AVERAGE to be typed on the console typewriter in the form specified by FORMAT number 5. Note that this is the same format as was used during input. The typed number will have the same form as the original data and might appear as follows: 2.35

After typing the answer the program proceeds to the control statement STOP and halts. Line 11 is the End Card, indicating the end of the program.

II. ARITHMETIC

BASIC ELEMENTS

QUANTITIES

The XDS 900 Series FORTRAN II Compiler is concerned with two modes of numerical quantities: integer quantities and floating-point quantities.

Integer quantities are used to represent integers of magnitude less than 8,388,608.(1)

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}.(1)

CONSTANTS

Constants are numbers which appear in a source program in explicit numerical form. They may be integer or floating-point.

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

EXAMPLES:

1

1973

32768

0

Floating-point constants are represented by a string of digits which contains 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.

(1) These limit values are due to the internal representation of numbers in the XDS 900 Series Computers. Integers are represented as 24 bit, two's complement binary numbers. Floating-point numbers are represented as a 39 bit two's complement mantissa and a 9 bit two's complement characteristic.

EXAMPLES:

3.14159265359

.004579

1.

0.

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 ten by which the floating-point constant is to be multiplied. The magnitude of the resulting number must be between the limits of 10^{-77} and 10^{77} or be zero.

The scale factor constant may be preceded by a "+" or "-" sign to indicate positive or negative powers of ten. If the sign is omitted, the power is considered positive.

EXAMPLES:

1.E-18	means	10^{-18}
.0271828E+2	means	2.71828
1.973E3	means	1973.

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

EXAMPLES:

5E-2	means	.05
1E+76	means	10^{76}
25E3	means	25000.

IDENTIFIERS

Identifiers are used to name variables, subprograms, and dummy arguments of subprogram definitions. An identifier is a string of letters and digits. Identifiers may be of any length; however, only the first eight characters will be used. The first character of the string must be a letter.

EXAMPLES:

M

DISCRIMINANT

CL2RW7

VARIABLES

Variables represent quantities which may take on a number of values and are referred to by name. They 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.

Variables may be scalar variables or array variables.

Scalar Variables

Scalar variables represent a single quantity and are denoted by scalar identifiers.

EXAMPLES:

Integer scalar variables

N

INDEX

K12

Floating-point scalar variables

SIGMA

X1

ERROR

Array Variables

An array variable represents a single element of an array of quantities rather than a single quantity. An array variable is denoted by the identifier (the array identifier) which names the array, followed by a subscript list enclosed in parentheses. The subscript list is composed of arithmetic expressions (see page 15) separated by commas.

EXAMPLES:

Integer array variables

```
K (3)
K (N+1)
MOVE (-1, M)
```

Floating-point array variables

```
X (N, 1, 1, M)
VOLTAGE (2* N+1, L, L+1)
```

Each expression gives the value of the corresponding subscript. The number of subscript expressions must equal the number of dimensions of the array.

Any expression may be used as a subscript. In particular, subscripting may be cascaded. If the value of a subscript 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 not less than the minimum and not greater than the maximum specified for the array.

EXAMPLES:

```
X (M* N+M-1)
K (THETA)
BRANCH (1+MOD(BRANCH (2*NODE), 3))
```

FUNCTIONS

Functions are subprograms which are referenced as basic elements in arithmetic expressions. 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.

FORM: identifier (argument, argument, . . . , argument)

An argument may be an arithmetic expression or an array identifier.

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. Functions not so identified are considered to be floating-point functions.

The mode of a function is independent of the modes of its arguments, i.e., an integer function may have floating-point arguments, etc.

EXAMPLES:

```
SIN (2* PI* TIME)
DOLLARS (PRICE)
MOD (M, K)
```

Functions 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.

Many library functions are included in the XDS FORTRAN System. These include elementary functions such as SIN, SQRT, etc. and arithmetic functions such as ABS, MOD, etc.

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)
SQRT (ALPHA)
```

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

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

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

```
(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

Operations of equal precedence are grouped from left to right. For example, the expression

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

is interpreted:

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

while the expression:

```
A/B/C/D
```

is interpreted to mean

```
( (A/B)/C)/D
```

and

A**B**C

is interpreted to mean

(A**B)**C.

Similarly,

A/B*C

is interpreted to mean

(A/B)*C

and

A/B*C/D

is interpreted

((A/B)*C)/D

Permutable sequences of operations will be reordered, if necessary, to increase object program efficiency.

EVALUATION

The numerical value of an expression may be of integer or floating-point mode. The mode of an expression is determined by the modes 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 XDS 900 Series FORTRAN II.

Integer Expressions

An integer expression is evaluated using integer arithmetic throughout, giving an integer value as the result. All results will be reduced modulo 2^{24}. Fractional parts arising in division are truncated, not rounded. For example, 5/2 yields 2; 2/3 yields 0.

EXAMPLES:

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

Floating-Point Expressions

Floating-point expressions are evaluated using floating-point arithmetic throughout, yielding a floating-point value. All results are limited in magnitude to the range 10^{-77} to 10^{77} or zero.

EXAMPLES:

```
(X(N-1)+X(N+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)
```

STATEMENTS

Assignment Statement

The assignment statement specifies an expression to be evaluated and a variable, called the statement variable, 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 assignment statement meaning "take the value of X, add one, and assign the resulting value to X."

EXAMPLES:

```
X    = X+1
K    = N*(L-1)
Y(M) = SINF(.06*M)
SUM  = SUM+TERM*X/N
```

The value of the expression in an assignment statement is made to agree in mode with the statement variable when the replacement is performed. Thus, an integer expression value is converted to a floating-point value if the statement variable is a floating-point variable, and a floating-point expression value is truncated to an integer if the statement variable is an integer variable.

For instance, in the statement

```
Z = N*(N-1)
```

the integer value of the expression is converted to floating-point before assignment to Z.

CALL Statement

FORMS: CALL identifier

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

This statement is used to call, or transfer control to, a subroutine subprogram. The identifier is the name of the subroutine.

The arguments, as in the case of functions, may be given as arithmetic expressions or array identifiers. Unlike a function, however, a subroutine may have more than one result and may use one or more of its arguments to return these results to the calling program. A subroutine may require no arguments at all, in which case the first form of the CALL statement is used.

EXAMPLES:

```
CALL DUMP
CALL FACTOR (A+1, 2*COS(THETA)*B(K), R1, R2)
CALL DOT (M, Y, Y, LENGTH)
```

The name of the subroutine has no bearing on the mode of its results. For instance, in the last example above the integer variable LENGTH might be the result of the subprogram DOT.

III. CONTROL

The normal flow of a FORTRAN program is sequential 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.

Statement Numbers

A statement number consists of an unsigned integer constant of five digits or less. Leading zeros are ignored; for example, 0002 and 2 are considered identical.

Although statement numbers appear in the source program as integers, they are not to be confused with numerical quantities. They represent a distinct type of basic quantity, viz. labels. Labels are used for identification of addresses in the object program.

Since statement numbers are used for identification, they must be unique; that is, no two statements may have the same number. No order or sequence is implied by the magnitudes of the statement numbers. Non-referenced statements need not be numbered; in fact, unnecessary numbering is wasteful of compiler storage.

Unconditional GO TO Statement

FORM: GO TO n

where n is a statement number.

This statement transfers control to the statement numbered n.

EXAMPLES:

```
GO TO 15
GO TO 957
```

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 (M(K) - JOB) 5, 2, 4
IF (Y) 14, 15, 15
```

In the first example above, control is transferred to statement 5 if M(K) < JOB, to statement 2 if M(K) = JOB, and to statement 4 if M(K) > JOB.

DO Statement

The DO statement allows a series of statements to be executed repeatedly under control of a variable whose value can change between repetitions and which may be integer or floating-point.

FORMS:

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

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

where n is a statement number.

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. The scalar variable of the DO statement is called the index. The values of $expression_1$, $expression_2$, and $expression_3$ are called, respectively, the initial, limit, and increment values of the index. If $expression_3$ is not stated (first form), it is understood to be 1.

Initially, the statements of the range are executed with the initial value assigned to the index. This initial execution is always performed, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value

of the index. In case the increment value is negative, another execution will be performed if the new value of the index is not less than the limit value.

After the last execution, control passes to the statement immediately following statement n. Exit may also be effected by a transfer from within the range of the DO statement.

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.

The index of a DO statement is treated as any other scalar variable. It is available for use within the range of the DO statement and outside of the range. The value of the index may be changed within the range of the DO statement. Similarly, the limit and increment values of the DO statement may be altered within the range of the DO statement.

A zero value of the increment is considered positive.

It is permissible to transfer into the range of a DO statement from outside of its range.

EXAMPLES:

```
      DO 2 L = 1, N
      DO 5 V = END, START, - .025
```

As an illustration of the use of DO statements, consider the sequence below.

```
      DO 4 L = 0, M
      CORR (L) = 0
      DO 3 K = L, N-1
3     CORR (L) = CORR (L) +X(K-L) *X(K)
4     CORR (L) = CORR (L) / (N-L)
```

Given that X and CORR are suitably specified arrays and that N>M>0, these statements will evaluate the autocorrelation function:

$$\text{CORR}(L) = \frac{1}{N-L} \sum_{K=L}^{N-1} X(K)\, X(K-L)$$

The summation is performed by the "inside" DO statement whose range ends with statement 3. The "outside" DO statement performs the division and changes the value of L.

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.

Computed GO TO Statement

The computed GO TO statement allows transfer of control to one of a group of statements, the particular statement chosen depending on conditions at run time.

FORM: GO TO (n_1, n_2, n_3, ..., n_k), expression

where n_1 n_2, ..., n_k are statement numbers. The comma preceding the expression is optional.

This statement transfers control to statement n_1, n_2, ..., n_k depending on whether the expression has the value 1, 2, ..., k respectively.

EXAMPLE:

GO TO (7, 12, 3, 4), K+1

will transfer control to the statement numbered 12 if K has the value 1.

The value of the expression is truncated to an integer if required. Expression values outside the range 1, 2, ..., k cause a run time error indication.

EXAMPLE:

GO TO (13, 27, 1, 4, 6), V (J)

This statement transfers control to statement 6 if V (J) has the value 5.728. A value of .57 for V (J) causes an error indication.

Assigned GO TO Statement

FORM: GO TO variable

This statement transfers control to the statement whose number was last assigned to the variable by an ASSIGN statement. The variable must appear in some previously executed ASSIGN statement.

EXAMPLES:

GO TO L

GO TO EXIT (3)

The variable of an assigned GO TO statement is a control variable and has a label as a value, not a numerical quantity. A control variable may be shared between a program and its sub-programs, like any other variable. The variable may appear in an ASSIGN statement in one program and be used in an assigned GO TO in another program.

ASSIGN Statement

FORM: ASSIGN integer TO variable

This statement sets the value of the variable for a subsequent assigned GO TO statement. The integer is the number of the statement to which control will be transferred by the assigned GO TO statement.

EXAMPLES:

```
ASSIGN 6 TO L
ASSIGN 72 TO EXIT(3)
```

As an example of the use of the ASSIGN and assigned GO TO statements consider the sequence below.

```
25      X = R*COS (THETA) *SIN (PHI)
        Y = R*SIN (THETA) *SIN (PHI)
        Z = R*COS (PHI)
        GO TO EXIT
```

This sequence may be used as a subroutine by other parts of the program. For instance the statements:

```
        ASSIGN 7 TO EXIT
        GO TO 25
7       . . .
```

will cause the subroutine to be executed and control returned to statement 7.

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 is checked at run time and must be one of the integers 0, 1, 2, ..., 24. A zero value causes all bits to be set to zero.

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 and 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, which is checked at run time, must be one of the integers 1, 2, ..., 24. The bit is set to zero after the test.

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 SET or RESET. The particular sense switch used is specified by the value of the expression, truncated if necessary. The resulting integer, checked at run time, must be 1, 2, 3, or 4.

EXAMPLES:

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

IF FLOATING OVERFLOW Statement

FORM: IF FLOATING OVERFLOW n_1, n_2

where n_1 and n_2 are statement numbers.

This statement tests for overflow on floating-point arithmetic operations. Arithmetic overflow occurs when the exponent of a floating-point result is out of bounds or a floating-point zero division is attempted. If the indicator is one, indicating that an overflow has occurred, control is transferred to statement n_1. If no overflow has occurred, control passes to statement n_2. The overflow indicator is initialized to zero and is set to zero after an IF FLOATING OVERFLOW statement.

EXAMPLE:

```
IF FLOATING OVERFLOW 15, 32
```

PAUSE Statement

FORM: PAUSE

PAUSE integer

This statement halts the machine. The integer and the location of the PAUSE are displayed. Program execution may be resumed from the computer console.

EXAMPLE:

```
PAUSE 62
```

STOP Statement

FORM: STOP

This statement causes termination of the program and returns control to the system.

RETURN Statement

FORM: RETURN

This statement returns control from an external 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 any point in the subprogram at which it is desired to terminate execution. Any number of RETURN statements can be used.

IV. INPUT-OUTPUT

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, etc.

In general, an input-output statement must provide:

1. Specification of the operation required.

   TYPE

2. The statement number of a FORMAT statement which will specify the format of the data and the sort of conversions required between the internal and external forms of the data, e.g.,

   TYPE 6

3. A list of the variables whose values are being transmitted. 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.

For example, the statement

TYPE 6, ALPHA, BETA, GAMMA

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

Similarly, the statement

ACCEPT 4, WINE, WOMEN, SONG

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

INPUT-OUTPUT RECORDS

All information appearing on external media (such as punched cards, magnetic tape, etc.) is grouped into records. The maximum amount of information allowed in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on the console typewriter, a record is one line; and so forth. The actual amount of information contained in each record is specified by the FORMAT statement.

Each execution of an input or output statement initiates the transmission of a new data record. Thus the statement

```
READ 2, EIN, ZWEI, DREI
```

is not necessarily equivalent to the statements

```
READ 2, EIN
READ 2, ZWEI
READ 2, DREI
```

since, in the second case, at least three separate records (in this case, punched cards) are required, whereas the single statement

```
READ 2, EIN, ZWEI, DREI
```

may require one, two, three, or more records depending upon format 2.

If an input-output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input-output statement. For instance, in the case of punched cards, two READ statements cannot input information from the same card, nor can two PUNCH statements output information on the same card.

If an input-output statement requests more than one record of information, successive records are transmitted until the statement is complete.

INPUT-OUTPUT LISTS

The list portion of an input-output statement indicates the order of transmission of the variable values. On input, the new values of the listed variables may be used in subscript or control expressions for variables occurring later in the list. For example

```
ACCEPT 5, K, A(K+1), X, Y(K)
```

reads in a new value for K and uses this value in the subscripts of the variables A and Y.

Indexing similar to that used in DO statements is allowed in input-output lists for handling array variables. The variables to be transmitted are listed, followed by the index control, and the whole is enclosed in parentheses to act as a single element of the list:

(variable, variable, ..., index control)

The index control has the same form as in the DO statement:

scalar variable = $expression_1$, $expression_2$, $expression_3$

or

scalar variable = $expression_1$, $expression_2$

The rules for repetition are the same as those for the DO statement. For example, the statement

TYPE 8, (FORCE (J), J = 1, 3)

is equivalent to

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

Each group enclosed within parentheses acts as an element of the list and is taken in order. Thus the statement

READ 2, (X(K), Y(K), K = 1, 2)

is equivalent to

READ 2, X(1), Y(1), X(2), Y(2)

but the statement

READ 2, (X(K), K = 1, 2), (Y(K), K = 1, 2)

is equivalent to

READ 2, X(1), X(2), Y(1), Y(2)

Indexing of this nature can be compounded in the same fashion as DO statements. For example

ACCEPT TAPE 2, ((TRIX(J, K), J = 1, 10), K = 1, 15)

means accept from paper tape a 10 by 15 matrix in the order

TRIX(1, 1), TRIX(2, 1), ..., TRIX(10, 1), TRIX(1, 2), ..., TRIX(10, 15)

If an entire array is to be transmitted, the indexing information may be omitted. The entire

array is transmitted in order of increasing subscripts with the first subscript varying most rapidly (i.e., columnwise). Thus, the above example can be written simply as

ACCEPT TAPE 2, TRIX

When more than one array is listed, the entire arrays are transmitted in the order they appear on the list.

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, J
```

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, K, (WESTCHESTER(L), L = 1, K)
```

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, (HIC, HAEC, HOC)
```

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 variables in the list. The data is converted from external to internal form as specified by FORMAT statement n.

EXAMPLE:

```
ACCEPT TAPE 17, B(J), J = 1, M
```

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, K, B(2, 1)
```

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, Z, (X(K), K = A, Z+1)
```

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(K, J ), K = 1, 10), J = 2, 14, 2)
```

Magnetic Tape Operations

Input-output statements which refer to magnetic tape units differ somewhat from those above. Since several tape units may be connected to the computer, the number of the tape unit required must be given by the input-output statement. This number is given by the value of an arithmetic expression, truncated to an integer if necessary. The tape number so specified is checked at run time for compatibility with the actual machine configuration.

Information may be transferred to or from magnetic tape in two forms; binary and BCD (Binary Coded Decimal). The binary form, used primarily for intermediate storage purposes, involves no data conversion and therefore no FORMAT statement reference.

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 is 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.

EXAMPLE:

```
READ INPUT TAPE 3, 5, A
READ INPUT TAPE K, 5, (A(J), B(J), J = 1, 10)
```

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 is equal to the value of the expression, truncated if necessary.

EXAMPLE:

```
READ TAPE 3, A, B
READ TAPE K, A, B, C(4,4)
```

WRITE OUTPUT TAPE Statement

FORM: WRITE OUTPUT TAPE expression, n, list

This statement causes the values of the 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(J), B, J = 1, 10)
```

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 is 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 is equal to the value of the expression, truncated if necessary.

EXAMPLES:

```
BACKSPACE 3
BACKSPACE K(N)
```

REWIND Statement

FORM: REWIND expression

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

EXAMPLE:

```
END FILE 3
```

FORMAT Statement

All input or output activity involving conversion of data requires the use of a FORMAT statement to specify the external format of the data and the type of conversion to be used. Any FORMAT statement can be used with any input-output medium (magnetic tape, paper tape, console typewriter, etc.).

FORMAT statements are not executed and may be placed anywhere in the program.

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

where S is a data field specification.

The separating commas may be omitted if no ambiguity results.

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

could be typed on the console typewriter.

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

.317250+2

.317250E2

.042739-45

31064

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

During input, the appearance of a decimal point "." in an E or F type number overrides the d specification of the field. In the absence of an explicit decimal point, the point is positioned d places from the right of the field, not counting the exponent, if present.

For example, a number with external appearance 271828E-1 and specification E12.5 is interpreted as 2.71828E-1.

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)
```

corresponds 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 holds for all E and F type conversions following the scale factor within the format statement. 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. The alphanumeric characters are transmitted as the value of a variable of an input-output list.

For example the statements

```
          READ 2, X
2         FORMAT (A5)
```

cause five characters to be input from a punched card and placed in memory as the value of the variable X.

Although w may have any value, the maximum number of characters transmitted is determined by the space allotted for the value of the variable. For an integer variable, the maximum is four characters; for a floating-point variable, the maximum is eight. Characters beyond the maximum are lost on input and replaced with blanks on output. A field width of less than the maximum causes blanks to be filled in after the given characters until the maximum is reached. That is, the characters are left justified.

Alphanumeric Format Fields

Alphanumeric fields may be specified within a FORMAT statement by simply enclosing the alphanumeric string in dollar signs "$".

For example, the statement

```
FORMAT ($ TEST COMPLETE $)
```

can be used to type

```
TEST COMPLETE
```

on the console typewriter..

The characters of an alphanumeric format field are not transmitted as values of variables. The characters are stored in the memory space allotted to the format specification itself and are transmitted to and from this space during input and output. The alphanumeric field is allotted space sufficient for exactly the number of characters, k, appearing between the dollar signs. An input-output list is not required for transmission of this type of field. During input, k characters are extracted from the input record and replace the k characters included within the specification. During output, the k characters specified, or the k characters which have replaced them, become part of the output record. For example, the statements

```
      ACCEPT 2
2     FORMAT ( $ TEST COMPLETE $)
```

can be used to replace the 15 characters TEST COMPLETE with the 15 characters NONCONVERGENT from the console typewriter.

Then the statement

```
TYPE 2
```

will type

```
NONCONVERGENT
```

An alternate method of specifying alphanumeric format fields is allowed. In this method, the alphanumeric string is preceded by the form kH, where k is the number of characters in the string. Blanks are counted. For instance, the format in the example above can be written:

```
FORMAT (15H TEST COMPLETE )
```

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 ($ VELOCITY = $,F8.4)

can result in the output line

VELOCITY = 6.4142

An alphanumeric 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. k must not equal zero.

Consider:

FORMAT ($TIME$, F8.4, 12X, X, F8.2)

This statement can be used to output

TIME 1.2863 X -148.61

where twelve blanks separate the two quantities.

Commas in Input Records

On input the occurrence of a comma within a numerical field causes termination of the field. This allows simplified preparation of data records using commas as field terminators For example, a data record using the format

FORMAT (I9, 4F12.6)

may be punched

5, 3.14, 7.2, 16.5, 9.34

The field width specified in the format must be greater than the number of characters encountered before the comma.

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 records.

The number of repetitions must not be zero.

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)

Alphanumeric fields can be repeated also in this manner. Thus

FORMAT (2($ AZIMUTH $))

is equivalent to

FORMAT ($ AZIMUTH $, $ AZIMUTH $)

Nesting of group repetitions is not allowed. The number of repetitions must not be zero.

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 slash "/" is used to indicate a new record. Thus, the statement

FORMAT (2F6.4/I3, 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 slash and the sequence of closing parentheses at the end of a FORMAT statement indicate the termination of a record.

For example the statements

```
      PUNCH 2, (A(K), K = 1, N)
2     FORMAT (E15.6)
```

cause the values of A(1), A(2), etc. to be punched one value to a card.

However, if

```
2     FORMAT (3E15.6)
```

is used, the values are punched three to a card. All values have format E15.6.

The total of all field widths specified for any record is the length of the record. If the record length specified is greater than the maximum allowed on a particular device, the excess characters are lost.

Maximum record lengths are:

1. Typewritten line - 80 characters
2. Punched card - 80 characters
3. Paper tape - 80 characters
4. Magnetic tape - 132 characters
5. Printed line - 132 characters

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

Printer Carriage Control

The first character of every line of information output to the line printer controls the movement of the paper form through the printer as follows:

Character	Action
blank	Single space
0	Double space; character is not printed
1	Eject to top of form; character is not printed
+	Suppress spacing; character is not printed
any other	Single space; character is printed

V. DECLARATIONS

A declaration is a description of certain properties of the program, rather than a specification of computation or other action. Several FORTRAN statements are used solely for the purpose of supplying the system with declarative information. These statements are primarily concerned with the interpretation of identifiers occurring in the source program and memory allocation in the object program.

CLASSIFICATION OF IDENTIFIERS

Each identifier appearing in a source program is classified as to the language element it identifies. Four main classifications are recognized:

scalar identifiers
array identifiers
subprogram identifiers
dummy identifiers

The classification is made according to the context in which the identifier first physically appears in the source program. This first appearance amounts to a declaration, explicit or implicit, of the proper interpretation of the identifier throughout the program.

DIMENSION Statement

The DIMENSION statement is used to declare an identifier to be an array identifier and to specify the number and limits of the array subscripts. Any number of arrays may be declared in a single DIMENSION statement.

The information provided by a DIMENSION statement is required for allocation of storage for arrays. Each array variable appearing in a program must represent an element of an array declared in a DIMENSION statement. The array variable must have the same

number of subscripts as were declared for the array and the value of each subscript must be within the limits specified by the DIMENSION statement. The DIMENSION statement must precede the first appearance of the array variable in the program.

FORM: DIMENSION S_1, S_2, ..., S_k

where S is an array specification.

Each array specification gives the array name and the minimum and maximum values each of its subscripts may assume, thus:

name (min/max, min/max, . . . , min/max)

The minima and maxima must be integers, signed or unsigned. The minimum must not exceed the maximum. Thus both negative and zero subscripts are permitted.

For example, the statement

DIMENSION X(-1/6, 2/5)

specifies X to be a two-dimensional array with the first subscript varying from -1 to 6, inclusive, and the second from 2 to 5, inclusive.

Minimum values of 1 may be omitted. For instance

DIMENSION Y(3, 4, 2)

is taken to be

DIMENSION Y(1/3, 1/4, 1/2)

EXAMPLES:

```
DIMENSION X(10)
DIMENSION X(10), Y(5, 0/6)
DIMENSION INDEX (-12/-8,7/20), ARG (400), FUN (2, 2, 2, 2)
```

SUBPROGRAM DEFINITION STATEMENTS

The subprograms which may be called, or referred to, by a FORTRAN program are classified as external or internal subprograms.

Internal subprograms are defined within the calling program. The definition is accomplished in a single statement-the arithmetic function definition statement. These subprograms

are defined, and referrable, only within the program containing the definition.

External subprograms are defined outside of the program which refers to them and are complete programs conforming to all the rules of FORTRAN programs. They may be compiled independently or with the main program which refers to them. The library programs included in the system are external subprograms.

Two types of external subprograms are available: the FUNCTION subprogram and the SUBROUTINE subprogram. The use of the declarations FUNCTION and SUBROUTINE in the definition of external subprograms is described in the sequel.

A subprogram, internal or external, may call other subprograms during its execution; however, recursion is not permitted.

DUMMY IDENTIFIERS

Subprogram definition statements declare certain identifiers to be dummy identifiers. These identifiers represent the arguments of the subprogram. When used in the subprogram they indicate the sort of elements which may appear as arguments and how the arguments are to be used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

Arithmetic Function Definition Statement

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

This statement serves to define an internal function for use in a particular program. The entire definition is contained in the single statement and this definition holds only in the program containing it. The appearance of a function name in an expression suffices to call the function during the evaluation of the expression at run time. The function has a single value whose mode is determined by the function identifier.

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

The list of identifiers enclosed in parentheses represents the argument list of the function. These identifiers 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. These identifiers must agree in order, number, and mode with the actual arguments presented to the function at run time. The number and mode of the arguments are checked at run time.

All arithmetic statement functions must precede the first executable statement of the program.

An argument of the function is specified in the defining expression through use of its corresponding identifier. Expressions are the only permissible arguments of internal functions; therefore the dummy identifiers may appear only as scalar identifiers in the defining expression. They may not appear as array subprogram identifiers.

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

EXAMPLE:

```
NUMBER (K) = K* (K+1)/2
BINV (X, B) = (X+B/X)/2
CATEN (X) = A(X)* BINV (EXP(X/A(X), 1)
```

In the last example, X is a dummy identifier and A(X) is a parameter.

FUNCTION Subprograms

A FUNCTION subprogram is a program which, as a function of one or more arguments, computes and returns a single result, and is called or referred to by the appearance of its name in an expression. A FUNCTION subprogram begins with a FUNCTION declaration and returns control to the main program by means of one or more RETURN statements.

FORM:

```
FUNCTION Identifier (identifier, identifier, . . . )
        .
        .
RETURN
        .
        .
RETURN
```

FUNCTION Statement

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

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

Identifiers appearing on the list enclosed in parentheses are dummy identifiers representing the function arguments. They must agree in order, number, and mode with the actual arguments presented to the function at run time. The number and mode of the arguments are checked at run time. FUNCTION subprogram arguments may be expressions or array names; therefore the dummy identifiers may appear as scalar or array identifiers. They may not appear as subprogram identifiers. Dummy identifiers which represent the names of arrays must appear in DIMENSION statements in the subprogram. Furthermore, the declared dimensions of each must equal the dimensions of the actual arrays presented to the function at run time.

A function must have at least one argument.

EXAMPLES:

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

As an example of a FUNCTION subprogram consider the following program which finds the inner product of two 3-dimensional vectors.

```
      FUNCTION DOT (V1, V2)
      DIMENSION V1 (3), V2(3)
      DOT = 0
      DO 2 K = 1, 3
2     DOT = DOT + V1 (k) * V2 (k)
      RETURN
```

The arguments of this function are floating-point array names, represented by the dummies V1 and V2. The value of the function is the single floating-point quantity DOT.

SUBROUTINE Subprograms

A SUBROUTINE subprogram differs from a FUNCTION subprogram in that it can be referred to only by a CALL statement and it may return more than one value. A SUBROUTINE subprogram begins with a SUBROUTINE declaration and returns control to the main program by means of one or more RETURN statements.

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

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 parentheses are dummy identifiers representing the arguments of the subroutine. These identifiers must agree in order, number, and mode with the actual arguments presented to the subroutine at run time. The number and mode of the arguments are checked at run time. SUBROUTINE subprograms may have expressions or array names as arguments, so the dummy identifiers may be used as scalar or array identifiers. Dummy identifiers may not be used as subprogram identifiers.

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

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

A SUBROUTINE subprogram need not have any arguments at all.

EXAMPLES:

```
SUBROUTINE SORT
SUBROUTINE FACTOR (COEF1,COEF2,COEF3,ROOT1,ROOT2)
SUBROUTINE NORMALIZE(X, K)
```

The program below is an example of a SUBROUTINE subprogram which finds the outer product of two 3-dimensional vectors.

```
SUBROUTINE CROSS (V1, V2 , V3)
DIMENSION V1(3), V2(3), V3(3)
V3(1) = V1(2)*V2(3)-V1(3)*V2(2)
V3(2) = V1(3)*V2(1)-V1(1)*V2(3)
V3(3) = V1(1)*V2(2)-V1(2)*V2(1)
RETURN
```

Notice that the name CROSS plays no part in the answer. The dummy array V3 is used to return the result.

IMPLICIT DECLARATION

Identifiers appearing in declaration statements such as DIMENSION, FUNCTION and SUBROUTINE are explicitly classified. If the first appearance of an identifier is not in a declaration, but in some imperative statement, the identifier is classified according to its context, i.e., implicitly. Examples of both types of declaration are shown in the program below. The program is a FUNCTION subprogram which finds the greatest distance from the origin attained by a set of points. The numbers marked with a # are line numbers for reference.

```
#1        FUNCTION  BIG RADIUS (X, Y)
#2        DIMENSION X(100), Y(100)
#3        R SQUARED (K) = X(K)*X(K)+Y(K)*Y(K)
#4        BIG R SQUARED = R SQUARED (1)
#5        DO 2 K = 2, 100
#6        IF (BIG R SQUARED - R SQUARED (K)) 1, 2, 2
#7  1     BIG R SQUARED = R SQUARED (K)
#8  2     CONTINUE
#9        BIG RADIUS  = SQRT (BIG R SQUARED)
#10       RETURN
```

Line # 1 declares BIG RADIUS to be the name of an external subprogram for use in other programs and to be the name of the scalar result of the subprogram. X and Y are declared to be dummy identifiers representing floating-point quantities. Line #2 further declares X and Y to represent array identifiers. Line #3 declares R SQUARED to be the name of an internal subprogram and K to be a dummy identifier representing an integer argument. This is because R SQUARED is used in a functional form and has not been previously declared to be a dummy, scalar, or array identifier. The identifier K is defined to be a dummy only within this single statement.

In line #4, BIG R SQUARED is implicitly declared to be a scalar floating-point variable. Line #5 implicitly declares K to be an integer scalar variable. Notice that the K of line #5 has no relation to the dummy K of line #3, and is not a dummy identifier. The line #5 K has meaning, and is used, else where in the program.

Line #9 implicitly declares SQRT to be an external subprogram name, since SQRT has not been previously declared a dummy, array, or scalar identifier and is being used as a function identifier.

MEMORY ALLOCATION

Memory allocation declarations supply the system with supplemental information regarding the storage of scalar variables and arrays.

COMMON Statement

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

The identifiers of a COMMON statement may be scalar or array identifiers. The COMMON statement specifies that the scalars 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 are 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.

Identifiers which correspond in this way must agree in mode for meaningful results.

EXAMPLES:

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

EQUIVALENCE Statement

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

FORM: EQUIVALENCE $(R_1, R_2, \ldots), (R_k, R_{k+1}, \ldots), \ldots$

where R denotes a location reference.

The location references of an EQUIVALENCE statement may be simple scalar or array identifiers or identifiers appended by a single integer constant enclosed within parentheses. The inclusion of two or more references in a parenthesis pair specifies that the quantities referenced share the same storage location. Such a group is called an equivalence set. For example

EQUIVALENCE (HOGAN, GOAT)

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

Quantities declared equivalent in this manner must be of the same mode.

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

DIMENSION CUBE (L_1/U_1, L_2/U_2, L_3/U_3)

where L and U denote the minimum and maximum values permitted for the subscripts. To calculate the location of the quantity

CUBE (K_1, K_2, K_3)

use the formula

$$\text{Location} = (K_3-L_3)*(U_1-L_1+1)*(U_2-L_2+1)+(K_2-L_2)*(U_1-L_1+1)+K_1-L_1+1$$

Thus, the statement pair

DIMENSION TEMP (10), CUBE (-1/1, -3/-2, 4)

EQUIVALENCE (TEMP(4), CUBE (7))

specifies that the quantities TEMP (4) and CUBE (-1, -3, 2) are to share the same storage location.

Notice that it is only the relative locations of the quantities within the array that matters, since the entire arrays are adjusted to satisfy the equivalence. In the example above, the statement

EQUIVALENCE (TEMP (2), CUBE (5))

has the same effect as

```
EQUIVALENCE (TEMP(4), CUBE (7) )
```

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
        L1:  ALPHA
        L2:  BETA
        L3:  GAMMA
        L4:  DELTA
```

then the statement

```
EQUIVALENCE (X, ALPHA (3) )
```

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

Note the 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.

Further Rules for COMMON and EQUIVALENCE

When quantities are involved in both COMMON and EQUIVALENCE statements, the COMMON statement takes precedence. Common storage is allocated first and quantities equivalenced to common quantities are placed as an overlay.

Any quantity which is to be stored in common must be placed there by a COMMON statement before being referenced by an EQUIVALENCE statement. Thus

```
COMMON  ALEPH
EQUIVALENCE  (ALEPH, BETH)
```

is a proper sequence, but

```
EQUIVALENCE(ALEPH, BETH)
COMMON ALEPH
```

is not allowed.

Furthermore, an equivalence is not assigned if it results in an overlay which extends beyond the bounds of common. For example, if A is a scalar and B an array of 5 quantities then

```
COMMON A
EQUIVALENCE (A, B (4))
```

would cause the array B to extend beyond the bounds of memory. The requested equivalence will not be assigned. However the sequence

```
COMMON B
EQUIVALENCE (A, B (4) )
```

is proper. The equivalence is the same, but common is four places larger.

No equivalence set may contain a reference to more than one quantity which previously has been equivalenced or placed in common. Thus the sequence

```
COMMON ONE
EQUIVALENCE (TWO, THREE)
EQUIVALENCE (ONE, TWO)
```

is not allowed. The equivalence may be accomplished correctly this way;

```
COMMON ONE
EQUIVALENCE (ONE, TWO, THREE)
```

or

```
COMMON ONE
EQUIVALENCE (ONE, TWO), (TWO, THREE)
```

APPENDIX A

```
      1  C      THE FOLLOWING STATEMENTS ILLUSTRATE FEATURES OF THE SDS 900 SERIES
=     2  C      FORTRAN II WHICH ARE NOT FOUND IN MANY OTHER FORTRAN COMPILERS.
=     3  C
=     4  C
=     5  C        CARD                    TECHNIQUES
=     6  C       NUMBER                   ILLUSTRATED
=     7  C
=     8  C          40 ARITHMETIC STATEMENT FUNCTION: NAME NOT ENDING IN F:
=     9  C             DUMMY VARIABLE SAME NAME: SUBSCRIPTED VARIABLE USED.
=    10  C          47 12-DIGIT ACCURACY.
=    11  C          52,53, 56-59 STATEMENT NUMBERS NOT RIGHT-JUSTIFIED.
=    12  C          55 MIXED MODE EXPRESSION.
=    13  C          56 SUBSCRIPTED SUBSCRIPTS-TO ANY LEVEL.
=    14  C          57 FLOATING POINT SUBSCRIPT EXPRESSIONS.
=    15  C          58 CONSECUTIVE EXPONENTIATION: MIXED 'E' FORM OF CONSTANTS.
=    16  C          59 IDENTIFIERS OF ANY LENGTH.
=    17  C          60 VARIABLES WITH SAME NAME AS FUNCTION.
=    18  C          62 SUBSCRIPTED VARIABLE ENDING IN F.
=    19  C          63 ANY NUMBER OF DIMENSIONS.
=    20  C          64 SUBSCRIPT RANGE SPECIFICATION, INCLUDING ZERO AND NEGATIVE.
=    21  C          66 BACKWARD DO LOOP: COMMA OPTIONAL: 5 DIGIT STATEMENT NUMBER
=    22  C          67 FIRST STATEMENT AFTER 'DO' NOT EXECUTABLE: $ USED TO DEFINE
=    23  C             ALPHANUMERIC STRING: COMMAS OPTIONAL WHERE UNAMBIGUOUS.
=    24  C          68 DO LOOP VARIABLE FLOATING: INDICES FLOATING EXPRESSIONS
=    25  C          69 NEGATIVE SUBSCRIPTS: MIXED MODE SUBSCRIPT EXPRESSIONS.
     26  C          72 DO LOOP VARIABLE CHANGED WITHIN LOOP.
-    27  C          73 DO LOOP LIMIT CHANGED WITHIN LOOP.
=    28  C          74 'END' OK AS BEGINNING OF STATEMENT.
=    29  C          75 INTEGERS UP TO 8,388,607 ALLOWED.
=    30  C          78 EXPRESSION PERMITTED INSTEAD OF CONSTANT:
=    31  C             TRANSFER INTO DO LOOP.
=    32  C          79 EXPRESSION PERMITTED INSTEAD OF VARIABLE: COMMA OPTIONAL.
=    33  C          80 FLOATING, SUBSCRIPTED VARIABLE IN 'ASSIGNED GO TO'
=    34  C          81 NO STATEMENT NUMBER LIST REQUIRED.
=    35  C          83 BACKWARD I/O LIST: PARENTHESES OPTIONAL.
=    36  C          88 EQUIVALENCE NOT PERMITTED TO ALTER COMMON.
=    37  C          89 'END' NOT LEFT JUSTIFIED.
=    38  C
=    39        DIMENSION ARRAY(10),MATRIX(4,4),LABEL(4),RETURN(5)
=    40        FUNCTION(FUNCTION) = ARRAY(I)*SQRT(FUNCTION)
=    41        I = 2
=    42        INTEGER = 83
=    43        N = 3
=    44        TIME = .5
=    45        DELTA X = .9
=    46        END = 4.0
=    47        A = .427/65345223
=    48        READ  27, LABEL,((MATRIX(J,K), K=1,4), J=1,4),ARRAY
=    49        PRINT 28, LABEL,((MATRIX(J,K), K=1,4), J=1,4)
=    50     12 FORMAT (//4F12.7/5F12.7//$SPACE TIME COORDINATE :$/6(8F10.5/)//
=    51       2         $ALPHA  =$4(3F10.5/8X)/1H1)
     52     27 FORMAT (20I4/10F8.4)
=    53     28 FORMAT (8LABEL  =$4I5//$MATRIX =$4(4I5/8X))
```

```
=     54      29 FORMAT (8F10.5)
=     55       1 A = 3*A + 4.0/I + ABSF(6*INTEGER)
=     56       2 B = ARRAY(MATRIX(LABEL(I),I))
=     57      3  C = ARRAY(-13.4 + SQRT(A+B))
=     58     4   D = B**C**(A-5E2)
=     59   5     SUM OF COEFFICIENTS = A+B+C+D
=     60         SIN = SINF(D)
=     61         DIMENSION SQRTF(5)
=     62         SQRTF(I) = 3
=     63         DIMENSION SPACE TIME COORDINATE (4,4,3,8)
=     64         DIMENSION ALPHA(-3/10,0/2,100/103)
=     65         READ 29, (((SPACE TIME COORDINATE (1,1,J,K), K=1,8), J=1,3)
=     66   54321 DO 7, J=N+I,N,-1
=     67      11 FORMAT (6HARRAY  =$10X10F10.4/)
=     68         DO 7 VELOCITY=ARRAY(J)/TIME,0,SIGNF(DELTA X,-ARRAY(J))
=     69       6 ALPHA(-1,J-N,100+VELOCITY/2) =
=     70        2    SPACE TIME(1,1,J-N+1,MIN(7,1.5*VELOCITY)+1)*J
=     71         IF (END-VELOCITY) 7,65,7
=     72      65 J = J-1
=     73         N = N-1
=     74       7 END = END + VELOCITY*12.0123456789
=     75         J = INTEGER-7654321
=     76         A = A-300
=     77         VELOCITY = VELOCITY/2
=     78         IF (SENSE SWITCH ABSF(A)/160) 6,8
=     79       8 GO TO (9,14) MOD (I,2) + 1
=     80       9 ASSIGN 10 TO RETURN(I)
=     81         GO TO RETURN(I)
=     82    14   STOP
=     83      10 PRINT 11,(ARRAY(I), I=10,1,-1)
=     84         PRINT 12, SUM OF COEFFICIENTS,SIN,END,VELOCITY,SQRTF,
=     85        2    (((SPACE TIME (1,J,K,A), A=8,1,-1), J=1,3,2), K=1,3,2*TIME),
=     86        3    ((ALPHA(-1,J,100-A), J=0,2), A=-3,0)
=     87         COMMON J, RETURN
=     88         EQUIVALENCE (RETURN,END)
=     89                E      N      D


COMMON ALLOCATION

  77777 J          77765 RETURN     77765 END

PROGRAM ALLOCATION

  00014 ARRAY      00040 MATRIX     00060 LABEL      00064 SQRTF
  00076 SPACETIM   01476 ALPHA      02216 I          02217 INTEGER
  02220 N          02221 K          02222 TIME       02224 DELTAX
  02226 A          02230 B          02232 C          02234 D
  02236 SUMOFCOE   02240 SIN        02242 VELOCITY

SUBPROGRAMS REQUIRED

  SQRT       ABSF       SINF       SIGNF      MIN        MOD


THE END
```

APPENDIX B

Compiler Diagnostics

The compiler does extensive error checking on FORTRAN source programs and pinpoints detected errors to facilitate correction. In general, errors are non-fatal; the object program may still be produced and run, bearing in mind the changes introduced by the errors, as described below.

Two types of diagnostics are provided by the compiler.

I. Statement Diagnostics

Most errors are caused by one particular statement being faulty. The compiler detects these errors at the time it encounters such a statement and prints an error indication beneath it on the listing. If the compiler is operating in the non-list mode, only the statements in error are listed, along with the error indications.

Statements in error are discarded and compilation then proceeds as if they had never existed.

The compiler proceeds from left to right in translating a source statement. When an error occurs, the compiler notes the character at which the error became evident and prints a △ underneath it on the listing. The delta may indicate an error of:

A. Omission - The statement has ended and something further is required. The △ will follow the last character in the statement, e.g.:

```
A=B**
    △
```

B. Commission - The flagged character does not make sense where it is. The compiler cannot proceed beyond it, e.g.:

```
A=SQRTF(/B)
        △
```

C. Usage - A number or identifier which is incorrect will be flagged underneath its last character, since it was at this point that the compiler had examined it completely, e.g.:

```
COMMON ALPHA, ALPHA
                △
```

An error message will also be printed on the following line. These messages are:

1. SYNTAX

   At the flagged character, the statement no longer conforms to the syntax of any recognized type of statement.

2. SUBSCRIPTS

   The number of subscripts being used with the array does not equal the number declared for the array.

3. ID DECLARATION

   The identifier marked is being used in a manner which contradicts a previous declaration.

4. ALLOCATION

   Allocation errors may occur in three statements:

   A. In a DIMENSION statement, either:
      1. A negative or zero dimension is specified.
      2. The lower limit for a subscript exceeds the upper limit.
      3. The requested size of an array exceeds 16K.

   B. An identifier appears in COMMON which has previously appeared in either COMMON or EQUIVALENCE.

   C. In an EQUIVALANCE set, more than one identifier has previously appeared in either COMMON or EQUIVALENCE.

5. NUMBER

   Number errors are of two types:

   A. The magnitude of the integer marked exceeds 8388607.

   B. The number marked is a statement label which does not fall between 1 and 99999 inclusive.

6. OVERFLOW

   The statement cannot be compiled due to either:

   A. Too many continuation cards.

   B. Exhaustion of the compiler's working storage. In this case, compilation is terminated and the compiler initializes for a new job.

II. Program Diagnostics

Certain errors cannot be detected until the entire source program has been read. These will be indicated beneath the source listing, with the summary listing. These are:

1. DO NEST ERRORS

   The statement numbers listed were meant to close the range of a DO statement. The compiler cannot close the DO loop correctly if:

   A. The closing statement is undefined. See under labeling errors.

   B. The closing statement is a transfer. The incrementing and testing of the DO loop will never take place.

   C. The closing statement is within the range of another DO statement which follows this one (i.e., the ranges partially intersect). The results of such a situation can be determined by inspection.

2. LABELING ERRORS

   The statement numbers listed are either:

   A. Undefined - The program will run normally until a transfer to one of these statements is actually attempted. At this point, the typeout "ERR LABL" will occur, and the program will not proceed.

   B. Multiply Defined - All transfers will be made to the last statement encountered with each of the particular numbers.

3. Errors Under COMMON ALLOCATION

   If the bounds of COMMON are exceeded by improper use of EQUIVALENCE, those variables which cannot be assigned as requested will appear under COMMON ALLOCATION, preceded by the word "ERROR" instead of an octal location. Such variables will then be assigned again under PROGRAM ALLOCATION as if they had never appeared in the EQUIVALANCE.

The following listing illustrates most of the different types of error diagnostics:

```
#      1  C      THE FOLLOWING STATEMENTS WILL ILLUSTRATE THE ERROR CHECKING
#      2  C      FEATURES OF THE SDS 900 SERIES FORTRAN II
#      3  C
#      4  C      ZERO OR NEGATIVE DIMENSIONS
#      5  C
#      6         DIMENSION ALPHA[0]
                                 Δ
ALLOCATION
#      7         DIMENSION BETA[-1,3]
                                 Δ
ALLOCATION
#      8  C
#      9  C      COMMON EXCEEDED [SEE BELOW UNDER COMMON ALLOCATION]
#     10  C
#     11         DIMENSION A[3],R[20]
#     12         COMMON X,Y,Z
#     13         EQUIVALENCE [A,Y]
#     14  C
#     15  C      FUNCTION NAME USED AS ARRAY
#     16  C
#     17     18 X # ROARING[I,B]
#     18        ROARING[20,20] # GOODOLD*GONEBY
                      Δ
ID DECLARATION
#     19  C
#     20  C      WRONG NUMBER OF SUBSCRIPTS
#     21  C
#     22         Y # A[I,J]
                        Δ
SUBSCRIPTS
#     23  C
#     24  C      NUMBER TOO LARGE
#     25  C
#     26         J # 123456789
                           Δ
NUMBER
#     27  C
#     28  C      ARRAY TOO LARGE
#     29  C
#     30         DIMENSION ENORMOUS[1000,1000]
                                          Δ
ALLOCATION
#     31  C
#     32  C      MISSING AND DUPLICATE STATEMENT NUMBERS [SEE BELOW]
#     33  C
#     34     13 X # Y
#     35     13 Y # X
#     36        GO TO 5
#     37  C
#     38  C      DO LOOP ERRORS [SEE BELOW]
#     39  C
#     40         DO 3 I#1,10
#     41         DO 4 J#1,3
#     42      4  IF [X-Y] 18,18,19
#     43     19  DO 6 I#1,10
```

```
#     44          DO 7 J#1,10
#     45        6 X # X&R[I]
#     46        7 Y # Y&R[I]
#     47   C
#     48   C      MISCELLANEOUS SYNTAX ERRORS
#     49   C
#     50          READ 41, [R[I], I#1]
                                  Δ
SYNTAX
#     51          X # 3.*[[2.&Y]*SQRT[3.14159265359/[Y**2&Z**2-4.7[P-Q]]] & ABS[P
                                                                        Δ
SYNTAX
#     52          X # ALPHA*BETA**[1.&SQRT[12.6*P*-Q]/3.5]-2.**J
                                                    Δ
SYNTAX
#     53          3.*P#Q
                  Δ
SYNTAX
#     54          IF [P-Q] 27,16
                             Δ
SYNTAX
#     55          X # -[1.&2.8*[R[3]-4.*R[I]*[3.-SQRTF[P&Q/[1.&X**2]]]]]]
                                                                          Δ
SYNTAX
#     56       14 FORMAT [4F12.5,I7,14HTOTAL VALUES F12.0]
                                                      Δ
SYNTAX
#     57          END


DO NEST ERRORS

      6         4         3

LABLING ERRORS

     13         5         3

COMMON ALLOCATION

  77776 X          77774 Y          77772 Z          ERROR A


PROGRAM ALLOCATION

  00005 A          00013 R          00063 I          00064 J
  00065 B

SUBPROGRAMS REQUIRED

  ROARING

THE END
```

APPENDIX C

SYNTAX

SYNTACTICAL DEFINITIONS

This section contains the precise definition of the syntactical structure of the XDS 900 Series FORTRAN II language. The definitions are those used by the processor itself and should serve as a reference for any question regarding syntax.

Restrictions on the definitions, such as maximum length of strings, lists and so forth, are not included here, nor is any reference to semantics. This information is found in the appropriate section of the text.

FORM OF THE DEFINITIONS

The definitions have the general form:

construct being defined:
    definition

The colon " : " means "is defined to be". The definitions usually contain other constructs which are defined elsewhere. The following elements are not defined as constructs and are considered basic:

integer
identifier
alphanumeric field

All constructs are written as one word or as a hyphenated sequence of words and are to be considered as indivisible symbols.

The metalinguistic symbols used are parentheses, quotation marks and the plus sign. The plus sign is read "or". Parentheses are used for grouping as in mathematical notation. Quotation marks are used to enclose literals. A plus sign followed by a blank means "or nothing".

As an illustration, consider the definition below.

```
call-statement:
    "CALL" identifier ("(" expression-list ")"  + )
```

This definition is read: "A call-statement is defined to be the letters CALL followed by an identifier. The identifier may be followed by a left parenthesis, an expression-list and a right parenthesis or the identifier may be followed by nothing at all".

Many of the definitions are recursive. For instance the definition:

```
expression-list:
    expression ("," expression-list + )
```

states that an expression list is a series of expressions separated by commas. Notice that at least one expression must be present.

APPENDIX D

SYNTAX FOR XDS 920/930 FORTRAN II

fortran-statement:

(arithmetic-statement + arithmetic-function-definition-statement + accept-statement + accept-tape-statement + assign-statement + assigned-go-to-statement + backspace-statement + call-statement + common-statement + computed-go-to-statement + continue-statement + dimension-statement + do-statement + end-file-statement + equivalence-statement + format-statement + function-statement + go-to-statement + if-statement + if-floating-overflow-statement + if-sense-light-statement + if-sense-switch-statement + pause-statement + print-statement + punch-statement + punch-tape-statement + read-statement + read-input-tape-statement + read-tape-statement + return-statement + rewind-statement + sense-light-statement + stop-statement + subroutine-statement + type-statement + write-output-tape-statement + write-tape-statement) end-of-statement

arithmetic-statement:

variable "=" expression

variable:

identifier ("(" expression-list ")" +)

expression-list:

expression ("," expression-list +)

expression:

("+" + "-" +) unsigned-expression

unsigned-expression:

term (("+" + "-") unsigned-expression +)

term:

factor (("*" + "/") term +)

factor:

primary ("**" factor +)

primary:

variable + function + constant + "(" expression ")"

function:

identifier "(" expression-list ")"

constant:

(integer ("." (integer +) +) + "." integer)

("E" signed-integer +)

signed-integer:

("+" + "-" +) integer

arithmetic-function-definition-statement:

identifier "(" identifier-list ")=" expression

identifier-list:

identifier ("," identifier-list +)

accept-statement:

"ACCEPT" integer ("," input-output-list +)

input-output-list:

(variable + "(" input-output-list ")") ("," (index-control + input-output-list) +)

index control:

identifier "=" expression "," expression ("," expression +)

accept-tape-statement:

"ACCEPT TAPE" integer ("," input-output-list +)

assign-statement:

"ASSIGN" integer "TO" variable

assigned-go-to-statement:

"GO TO" variable (",)" integer-list ")" +)

backspace-statement:

"BACKSPACE" expression

call-statement:

"CALL" identifier ("(" expression-list ")" +)

common-statement:

"COMMON" identifier-list

computed-go-to statement:

"GO TO (" integer-list ")" (", " +) expression

integer-list:

integer ("," integer-list +)

continue-statement:

"CONTINUE"

dimension-statement:

"DIMENSION" dimension-list

dimension-list:

identifier "(" limit-list ")" ("," dimension-list +)

limit-list:

((signed-integer "/" signed-integer) + ("+" +) integer) ("," limit-list +)

do-statement:

"DO" integer (", " +) index-control

end-file-statement:

"END FILE" expression

equivalence-statement:

"EQUIVALENCE" equivalence-list

equivalence-list:

"(" equivalence-set ")" ("," equivalence-list +)

```
equivalence-set:
        identifier ("(" integer ")" + ) ", " reference-list

reference-list:
        identifier ("(" integer ")"+ ) (", " reference-list + )

format-statement:
        "FORMAT (" (format-list + ) ")"

format-list:
        format-basic ( (", " + ) format-list + )

format-basic:
        "$" alphanumeric-field "$" + (signed-integer + ) "P" +
        (integer + ) ("/" + "X" + "H" alphanumeric-field +
        "(" format-list ")" + ("I" + "A") integer +
        ("F" + "E") integer "." integer)

function-statement:
        "FUNCTION" identifier "(" identifier-list ")"

go-to-statement:
        "GO TO" integer

if-statement:
        "IF (" expression ")" integer "," integer "," integer

if-floating-overflow-statement:
        "IF FLOATING OVERFLOW" integer "," integer

if-sense-light-statement:
        "IF (SENSE LIGHT" expression ")" integer "," integer

if-sense-switch-statement:
        "IF (SENSE SWITCH" expression ")" integer "," integer

pause-statement:
        "PAUSE" (integer + )

print-statement:
        "PRINT" integer ("," input-output-list + )
```

punch-statement:

"PUNCH" integer ("," input-output-list +)

punch-tape-statement:

"PUNCH TAPE" integer ("," input-output-list +)

read-statement:

"READ" integer ("," input-output-list +)

read-input-tape-statement:

"READ INPUT TAPE" expression "," integer ("," input-output-list +)

read-tape-statement:

"READ TAPE" expression ("," input-output-list +)

return-statement:

"RETURN"

rewind-statement:

"REWIND" expression

sense-light-statement:

"SENSE LIGHT" expression

stop-statement:

"STOP"

subroutine-statement:

"SUBROUTINE" identifier ("(" identifier-list ")" +)

type-statement:

"TYPE" integer ("," input-output-list +)

write-output-tape-statement:

"WRITE OUTPUT TAPE" expression "," integer ("," input-output-list +)

write-tape-statement:

"WRITE TAPE" expression ("," input-output-list +)

APPENDIX E

XDS 920/930 FORTRAN II Statements

1. Assignment 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. FUNCTION Statement
18. GO TO Statement
19. IF FLOATING OVERFLOW Statement
20. IF Statement
21. IF SENSE LIGHT Statement
22. IF SENSE SWITCH Statement
23. PAUSE Statement
24. PRINT Statement
25. PUNCH Statement
26. PUNCH TAPE Statement

27. READ Statement
28. READ INPUT TAPE Statement
29. READ TAPE Statement
30. RETURN Statement
31. REWIND Statement
32. SENSE LIGHT Statement
33. STOP Statement
34. SUBROUTINE Statement
35. TYPE Statement
36. WRITE OUTPUT TAPE Statement
37. WRITE TAPE Statement

INDEX