CONFIDENTIAL

PDP-X Technical Memorandum # 28

Title: PDP-X FORTRAN IV Initial Specifications

Author(s): J. Cohen

Index Keys: Software Specifications
FORTRAN

Distribution Key: B, C

Obsolete: None

Revision: None

Date: October 14, 1967

This memo is an initial attempt to define the PDP-X FORTRAN language. Much of what appears here has been taken directly from the PDP-9 FORTRAN manual. In general, an attempt has been made to make the PDP-X language contain at least as much as the PDP-9 language. I would appreaciate hearing from anyone who has any comments or suggestions for change.

John Cohen

digital INTEROFFICE MEMORANDUM

DATE:

SUBJECT: Initial Specifications for PDP-X Fortran Language

TO: Bill Segal FROM: John Cohen

1. Introduction

The purpose of this memo is to give an initial set of specifications for the fortran IV language we shall implement on the PDP-X. The main objective here is to indicate the valid syntax of the source statements. In general, the target language is not considered. The memo should be thought of as being directed to the systems programmers who will implement the compiler, rather than the compiler users.

Much of the information that appears here has been lifted in one form or another fro m the PDP-9 fortran manual. Also, the ASA specifications have been used to some degree. An appendix at the end of this memo indicates the differences between this specification and the ASA language.

1.1 PDP-X Fortran Statement Formats

First we shall consider the card format. Each card may be thought of as containing 4 fields - the statement number field (columns 1 through 5), the continuation field (column 6), the statement body field (column 7 through 72) and the identification field (column 73 through 80). Generally, statements will fit on a single card in the statement field (7 through 72). If it is necessary to use more than one card, succeeding cards have a character other than blank or zero in the continuation field (column 6). There may be as many as 5 continuation cards, or a maximum of 6 cards in any single statement. There will be no special restrictions on what must appear in the first card, as in the PDP-9 fortran.

Statement numbers consist of from 1 to 5 digits. Leading zeros and embedded blanks are effectively ignored. If the multi card format is used, any statement number must appear in the statement number field of the first card. This field must be blank in succeeding cards.

The compiler ignores columns 73 through 80 of all cards. This field conventionally is used for a program identification and sequence numbering. In addition, if a card has a C in column 1, the entire card will be ignored. This gives the programmer a means of interspersing comments within the source program.

The paper tape format is the same as the card format with a few exceptions. First, each statement line must be terminated by a carriage return, line feed sequence. If this

does not occur after the eightieth character, trailing blanks will be inserted. In order to make preparation of paper tapes easier, an initial tab, followed by an alpha character, will put the alpha character effectively in column 7. This device is a convenient means for skipping over the statement number field. Also, a tab, followed by a digit, will put the digit in column 6. This is a convenient means for starting succeeding cards in the multi card format. If the statement number ends before column 6, the tab punch, followed by an alpha character, will effectively put the alpha character in column 7.

1.2 Scope of Specifications

Section 2. of this memo describes the basic elements of the fortran language. These include the character set, constants, variables, and expressions.

Fortran statements specify a sequence of computations to carry out the processes of a program. An arithmetic statement defines a numerical calculation. A control statement determines the sequence of operations in a program. Input output statements control the transmission of information from the computer to external devices and vice versa. Specification statements define the properties of variables, control memory allocation and perform other similar functions. Sections 3 and 4 describe the arithmetic and control statements, respectively. Section 5 contains the input/output statements. These include the executable statements (READ, WRITE) and the nonexecutable FORMAT statement. The specification statements are described in section 6.

Section 7 describes the subprograms - functions, subroutines and block data. Finally, an appendix compares the PDP-X fortran with the ASA standards. This indicates both where the PDP-X fortran does some things not in the ASA standard and where it does not do some things specified in the standard.

2. Elements of the Fortran Language

Here we will discuss the character set and the data elements, constants and variables. We shall also discuss the various arithmetic and logical expressions and the rules for their evaluations.

2.1 Character Set

There are 48 characters used in the language. These consist of the 26 letters, the 10 digits and the following 12 special characters:

Blank	
Equals	=
Plus	+
Minus	-

Asterisk	*
Slash	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point	.
Dollar Sign	$
Quote	"

2.2 Data Elements

Before defining the data elements, it would be well to discuss some general concepts of interest to the implementer, if not the user. Data elements are contiguous blocks of storage which hold data accessed by the users program. The name of each data element can be thought of as a "variable name"specified by the user, or as a series of actual machine locations. The contents of the data element are the instantaneous values in the machine locations. Elements can be categorized as to size - the number of items in an element: type - the nature of each item in the data element (integer, real, double precision, etc.); and as to whether the element is a constant or a variable.

2.2.1 Data Element Types

Integers are stored one per PDP-X word (15 bits and sign). The range is from -32767 to +32767. Real numbers are stored in 2 PDP-X words. Bit zero contains the sign, bits 1 through 7 contain an exponent base 16, and bits 8 through 31 contain the 6 hex digit mantissa. Thus the accuracy of single precision real numbers is 5 hexdigits, or approximately 7 decimal digits.

Double precision numbers are stored in 4 PDP-X words. Bit zero contains the sign, bits 1 through 7 the exponent base 16, and bits 8 through 63 the mantissa in 14 hex digits. The accuracy is 13 hex digits or about 17 decimal digits.

Logical items are contained in one word This word contains zero to imply FALSE and one to imply TRUE.

My initial feeling is that we should use two characters per word for alphameric data. This would eliminate the need for much packing and unpacking and among other things would make implementation of the A format specifier convenient. If alphameric data does no appear in a user program to any great degree, what we would save with the smaller format interpreter would more than make up for the core we use to pack characters two per word rather than three. However, we may still want to use modulus 50 arithmetic to pack the data three per word and I think this should be left open for now.

2.2.2 Variable Data Elements

In a fortran program, variable data elements are referred to by variable names. These consist of from 1 - 6 letters or digits such that the first character is a letter. Some examples are:

LEGAL	ILLEGAL
X	123X (first character not a letter)
XMAX	MAXIMUM (too many characters)
X123	

Associated with each data name is a data type. This can be determined implicitly from the first character of the name or it can be determined explicitly by the specification statements (section 6). A data name is implicitly an integer if the first character is I, J, K, L, M or N. It is implicitly real if the first character is not I, J, K, L, M or N. The type specification statements override the implicit categorization and allow the user to establish data elements as integer, real, double precision real and logical. Thus integer and real elements can be established either implicitly or explicitly, but double precision real and logical must be established explicitly.

The fortran language has no separate data type for alphamerics. The user may put alphameric data into variables of any legal type, as long as he is aware of the number of characters he can get into items of each type. Assuming that we go 2 characters per word, this would be 2 characters per integer, 4 characters per real number, 8 characters per double precision real number and 2 characters per logical variable.

2.2.3 Arrays of Variable Data Items

A data element with a single data item is called a scalar. If a data element has more than one item, it is called an array. An array can be a simple sequence (or one dimensional). As an example, suppose that A is a one dimensional array with 10 items. Then, A(3) is the third item in the array A.

An array can be two dimensional (equivalent to a matrix). As an example, suppose that B is a two dimensional array with 5 rows and 10 columns. Then, D(2, 3) is the name of the item in the second row, third column of D. PDP-X fortran can have arrays of 1, 2 and 3 dimensions.

Arrays are arranged in storage in ascending absolute locations. The first subscript varies most rapidly, the second varies next most rapidly, etc. For example, a 3-dimensional array A, defined in a dimension statement (section 6) as a (2,2,2) will be stored sequentially in this order:

```
A(1,1,1)
A(2,1,1)
A(1,2,1)
A(2,2,1)
A(1,1,2)
A(2,1,2)
A(1,2,2)
A(2,2,2)
```

To refer to a particular item in an array the fortran programmer must specify as many subscripts as the array has dimensions. These subscripts are enclosed in parenthesis and separated by commas. Each subscript may be any integer expression (integer expressions are defined later in this section). Examples are:

```
A(I)
B(I, J - 3)
BETA(I, NA (3*1, J))
```

2,2.4 Constant Data Elements

Fortran constants have the following characteristics:

-- they are scalar data elements (not arrays).
-- their value remain fixed at all times
-- they are referred to by names which are "the same" as their contents.

What we shall discuss next is the method in which these constants are named We will use the term "constant" where we really mean "the name of the constant", but this should cause no confusion.

2.2.4.1 Integer Constants

An integer constant consists of from 1 to 5 digits, optionally preceded by a plus or minus sign. The magnitude of the constant must be equal or less than 32767. Examples are:

```
1
-5123
6177
```

2.2.4.2 Real Constants

Real numbers may be written in two forms. The first form begins with an optional plus or minus sign and is followed by a digit string containing 6 significant digits with a decimal

point in some position. The second form begins with an optional plus or minus sign, followed by from 1 to 6 significant digits, with or without an embedded decimal point, followed by the character E, an optional plus or minus sign and a 1 or 2 digit base 10 exponent. The magnitude of this exponent must be equal or less than 75. Examples are:

```
1.
-51.23
10. E-6
7932E21
```

2.2.4.3 Double Precision Real Constants

Double precision reals begin with an optional plus or minus sign, followed by from 1 to 6 significant digits, with or without an embedded decimal point, followed by the character D and a 1 or 2 digit exponent. The magnitude of the exponent must be equal or less than 75.

2.2.4.4 Logical Constants

The logical constants are represented as follows:

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

2.2.4.5 Alphameric Constants

Alphameric constants may be written as an unsigned integer constant N, greater than zero and less than 100, followed by the character H, followed by exactly N alphameric characters (including blank). An alternate method is to enclose the alphameric characters in quotation marks.

Alphameric constants may be used only in data statements (chapter 6) or CALL statements (chapter 7). Examples are:

```
10HABCDEFGHIJ
'ABCDEFGHIJ'
```

2.3 Expressions

An expression is a combination of elements (constants, subscripted on nonsubscripted variables, in functions) each of which is related to another by operators and parenthesis. An expression represents one single value which is the result of the calculations specified by the values and operators that make up the expression. The fortran language provides two kinds of expressions: arithmetic and logical.

2.3.1 Arithmetic Expressions

An arithmetic expression consists of arithmetic elements joined by the following operators:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

An expression may consist of a single element (meaning a constant, a variable, or a function name). An expression enclosed in parentheses is considered a single element. Compound expressions use arithmetic operators to combine single elements.

The type of quantities making up an expression determine its mode, i.e. a simple expression consisting of an integer constant or an integer variable is said to be in the integer mode. Similarly, real constants or variables produce a real mode of expression, and double precision constants or variables produce a double precision mode.

In general, variables or constants of 1 mode cannot be combined with variables or constants of another mode in the same expression. The following indicates the exceptions to this rule:

First Operand	Operator	Second Operand	Result
real	+ - * /	double	double
double	+ - * /	real	double
real	**	integer	real
real	**	double	double
double	**	integer	double
double	**	real	double

The order in which operations of an arithmetic expression are to be computed is based on a priority rating. The operator with the highest priority takes precedence over other operators in the expression. Parentheses may be used to determine the order of computation. If no parentheses are used the order is understood to be as follows:

-- function reference
-- exponentiation
-- multiplication
-- addition and subtraction

The rules for constructing arithmetic expressions are as follows:

-- any expression may be enclosed in parenthesis.
-- expressions may be preceded by a plus or minus sign.
-- simple expressions may be connected to other simple expressions to form a coupound expression, provided that no two operators appear together and no operator is assumed to be present.
-- only valid mode combinations may be used in an expression.
-- the expression must be constructed so that the priority scheme determines the order of operation desired.

2.3.2 Logical Expressions

The first logical expression we shall consider is a relational expression. This consists of two arithmetic expressions separated by a relational operator. The result value is either true or false, depending upon whether the condition expressed by the relational operator is met or not met. The arithmetic expressions may both be on the integer mode or they may be a combination of real and/or double precision. No other combinations are legal. The relational operators must be preceded and followed by a decimal point. They are:

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

Examples are:

```
I   .LT. IMAX
A+B .NE. 712.3
```

A general logical expression consists of logical elements joined by logical operators. The value is either true or false. The logical operator symbols must be preceded and followed by a decimal point. They are:

.NOT. Logical negation. Reverses the state of the logical quantity that follows.

.AND. Logical AND generates a logical result (TRUE or FALSE) determined by two logical elements as follows:
T .AND. T generates T
T .AND. F generates F
F .AND. T generates F
F .AND. F generates F

.OR. Logical OR generates a logical result determined by two logical elements as follows:
T .OR. T generates T
T .OR. F generates T
F .OR. T generates T
F .OR. F generates F

The following rules govern the construction of logical expressions:
-- A logical expression may consist of a logical constant, a logical variable, a reference to a logical function, a relational expression, or a complex logical expression enclosed in parenthesis.
-- The logical operator NOT need only be followed by a logical expression, while the logical operators AND and OR must be both preceded and followed by a logical expression.
-- Any logical expression may be enclosed in parenthesis. The logical expression following the logical operator NOT must be enclosed in parenthesis if it contains more than 1 quantity.
-- No two logical operators may appear in sequence, not separated by a comma or parenthesis, unless the second operator is NOT. In addition, no two decimal points may appear together, not separated by a comma or parenthesis unless one belongs to a constant and the other to a relational operator.

Parenthesis may be used as in normal mathematical notation to specify the order of operations. Within parenthesis, or where there are no parenthesis, the order in which operations are performed is as follows:
-- Arithmetic operations within the arithmetic expressions of relational expressions
-- The relational operators
-- NOT
-- AND, OR

3. Arithmetic Statements

An arithmetic statement is a mathematical equation written in the FORTRAN language which defines a numerical or logical calculation. It directs the assignment of a calculated quantity to a given variable. An arithmetic statement has the form

V = E

where V is a variable (integer, real, double-precision, or logical, subscripted or nonsubscripted) or any array element name; = means replacement rather than equivalence, as opposed to the conventional mathematical notation; and E is an expression.

In some cases, the mode of the variable may be different from that of the expression. In such cases an automatic conversion takes place. The rules for the assignment of an expression E to a variable V are as follows:

V Mode	E Mode	Assignment Rule
Integer	Integer	Assign
Integer	Real	Fix and assign
Integer	Double-precision	Fix and assign
Real	Integer	Float and assign
Real	Real	Assign
Real	Double-precision	Double-precision evaluate and real assign
Double-precision	Integer	Double-precision float and assign
Double-precision	Real	Double-precision evaluate and assign
Double-precision	Double-precision	Assign
Logical	Logical	Assign

Mode conversions involving logical quantities are illegal unless the mode of both V and E is logical. Examples of an assignment statements are:

```
ITEM = ITEM + 1
A(1) = B(1)
V = .FALSE.
X = A .GT .B .AND. C .LE. G
A = B
```

4. CONTROL STATEMENTS

The statements of a FORTRAN program normally are executed in the order they are written. However, it is frequently desirable to alter the normal order of execution. Control statements give the FORTRAN user this capability. This section discusses the reasons for control statements and the ways in which they may be used.

4.1 Unconditional Go To Statements

The form of the unconditional GO TO statement is

GO TO n

where n is a statement number. Upon the execution of this statement, control is transferred to the statement identified by the statement number, n, which is the next statement to be executed. Example:

GO TO 17

4.2 Assign Statement

The general form of an ASSIGN statement is

ASSIGN n TO i

where n is a statement number and i is a nonsubscripted integer variable name which appears in a subsequently executed assigned GO TO statement. The statement number, n, is the statement to which control will be transferred after the execution of the assigned GO TO statement. Example:

ASSIGN 27 TO ITEST

4.3 Assigned Go To Statement

Assigned GO TO statements have the form

GO TO i (n1, n2, ..., nm)

where i is an nonsubscripted integer variable reference appearing in a previously executed ASSIGN statement, and n1, n2, ..., nm, are the statement numbers which the ASSIGN statement may legally assign to i. Examples:

```
ASSIGN 13 TO KAPPA
GO TO KAPPA (1, 13, 72, 100, 35)
```

There is no object time checking to ensure that the assignment is one of the legal statement numbers.

4.4. Computed Go To Statement

The format of a computed GO TO statement is

GO TO (n1, n2, ..., nm), i

where n1, n21 ...., nm are statement numbers and i is an integer variable reference whose value is greater than or equal to 1 and less than or equal to the number of statement numbers enclosed in parentheses. Example:

GO TO (3, 17, 25, 50, 66), ITEM

If the value of ITEM is 2 at the time this GO TO statement is executed, the statement to which control is transferred is the statement whose number is second in the series, i.e., statement number 17.

4.5 Arithmetic If Statement

The form of the arithmetic IF statement is

IF (e) n1, n2, n3

where e is an arithmetic expression and n1, n2, n3 are statement numbers. The IF statement evaluates the expression in parentheses and transfers control to one of the referenced statements. If the value of the expression (e) is less than, equal to, or greater than zero, control is transferred to n1, n2, or n3 respectively. Example

IF (AUB (1) - B*D) 10, 7, 23

4.6 Logical If Statement

The general format of a logical IF statement is

IF (e) s

where e is a logical expression and s is any executable statement other than a DO statement or another logical IF statement. The logical expression is evaluated, and different statements are executed depending on whether the expression is true or false. If the logical expression e is true, statement s is executed and control is then transferred to the following statement (unless the statement s is a GO TO statement or an arithmetic IF statement, in which cases control is transferred as indicated; or the statement s is a CALL statement, in which case control is transferred to the next statement after return from the subprogram).

If the logical expression e is false, statement s is ignored and control is transferred to the statement following the IF statement. Examples are:

```
IF (L1) I = I + 1
IF (L .LE. k) GO TO 17
IF (LOG. AND. (.NOT. LOG 1) ) IF (X) 3,5,5
```

4.7 Do Statement

The DO statement is a command to execute repeatedly a specified series of statements. The general format of the DO statement is

```
DO n i = m1, m2, m3
```

or

```
DO n i = m1, m2
```

where n is a statement number representing the terminal statement or the end of the "range"; i is a nonsubscripted integer variable known as the "index"; and m1, m2, and m3 are integer expressions, which represent the "initial," "final," and increment" values of the index. If m3 is omitted, as in the second form of the DO statement, its value is assumed to be 1.

The DO statement is a command to execute repeatedly a group of statements following it up to and including statement n. The initial value of i is m1 (m1 must be less than or equal to m2). Each succeeding time the statements are operated, i is increased by the value of m3. When i is greater than m2, control passes to the statement following statement number n.

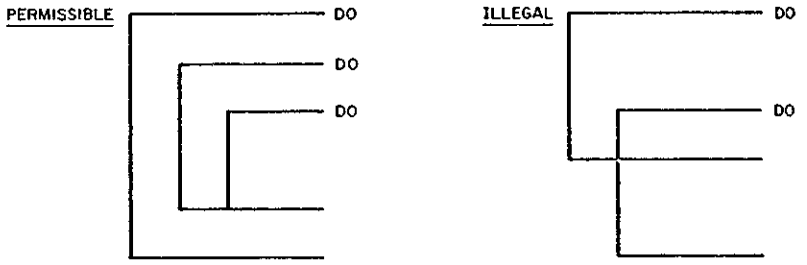
The range of a DO statement is a series of statements to be executed repeatedly. It consists of all statements immediately following the DO, up to and including statement n. Any number of statements may appear between the DO and statement n. The terminal statement (statement n) may not be a GO TO (of any form), an arithmeticIF, a RETURN, a STOP, a PAUSE, or a DO statement, or a logical IF statement containing any of these forms.

The index of a DO is the integer variable i which is controlled by the DO statement in such a way that its initial value is set to m1, and is increased each time the range of statements is executed by m3, until a further incrementation would cause the value of m2 to be exceeded. Throughout the range of the DO, the index is available for computation either as an ordinary integer variable or as the variable of a subscript. However, the index may not be charged by any statement within the DO range.

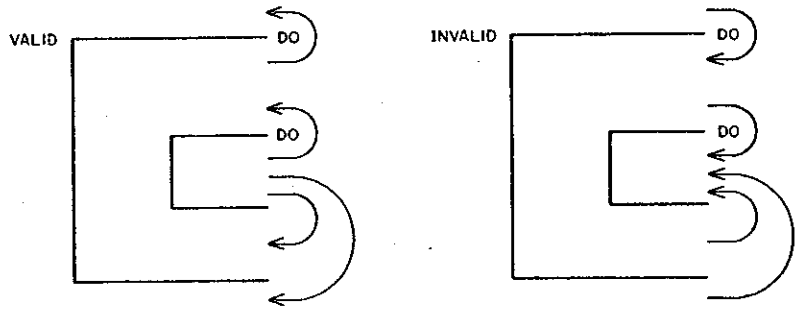
The initial value is the value of the index at the time the range is executed for the first time. The final value is the value which the index must not exceed. When the condition is satisfied the DO is completed and control passes to the first executable statement following statement n. The increment is the amount by which the index is to be increased after each execution of the range. If the increment is omitted, a value of 1 is implied. Example:

```
DO 72 I = 1, 10, 2
DO 15K = 1, 5
DO 23 I = I, K, NP (K + 1) + 1
```

Any FORTRAN statement may appear within the range of a DO statement, including another DO statement. When such is the case, the range of the second DO must be contained entirely within the range of the first; i.e., it is not permissible for the ranges of DOs to overlap. A set of DOs satisfying this rule is called a nest of DOs. It is possible for a terminal statement to be the terminal statement for more than one DO statement. The following configuration, where brackets are used to represent the range of the DOs, indicates the permissible and illegal nesting procedures.



Transfer of control from within the range of a DO statement to outside its range is permitted at any time. However, the reverse is not true; i.e., control cannot be transferred from outside the range of a DO statement to inside its range. The folloring examples show both valid and invalid transfers.



4.8 Continue Statement

The CONTINUE statement causes no action and generates no machine coding. It is a dummy statement which can be used for terminating DO loops when the last statement would otherwise be an illegal terminal statement (i.e., GO TO, arithmetic IF, RETURN, STOP, PAUSE, or DO, or a logical IF containing any of these forms). The form consists of the single word:

CONTINUE

4.9 Pause Statement

A PAUSE statement is a temporary halt of the program at run time. The PAUSE statement has one of the two forms:

PAUSE

or

PAUSE n

where n is an octal integer whose value is less than (77777)8. The integer is typed out on the console Teletype for the purpose of determining which of several PAUSE statements was encountered. Program execution is resumed by operator intervention, starting with the first statement following the PAUSE statement.

4.10 Stop Statement

The STOP statement is of one of the forms:

STOP

or

STOP n

where n is an octal integer whose value is less than (77777)8. The STOP statement is placed at the logical end of a program and causes the computer to type out on the console Teletype the integer n and then to exit back to the Monitor.

4.11 End Statement

The END statement is placed at the physical end of a program or subprogram. The form consists of the single word:

END

The END statement is used by the compiler and generates no code. It signals the compiler that the processing of the source program is complete.

5. Input/Output Statements

The input/output (I/O) statements direct the exchange of data between the computer and I/O devices. The information thus transmitted by an I/O statement is defined as a logical record, which may be formatted or unformatted. A logical record, or records, may be written on a device as one or more physical records. This is a function of the size of the logical record(s) and the physical device used. The definition of the data which comprises a user's optimum physical record varies for each I/O device, as follows:

Unit or Device	Formatted Physical Record Definition	Unformatted (Binary) Physical Record Definition
Typewriter (input and output)	One line of type is terminated by a carriage return. Maximum of 72 printing characters per line	Undefined
Line printer	One line of printing. Maximum of 120 characters per line	Undefined
Cards (input and output)	One card. Maximum of 80 characters	50 words
Paper tape (input and output)	One line image of 72 printing characters	30 words
Magnetic tape	One line image of 630 characters	252 words
Disc/drum/ DECtape	One line image of 630 characters	252 words

Each I/O device is identified by an integer constant which is associated with a device assignment table in the PDP-9 Monitor. This table may be modified at system generation time, or just before run time. For example, the statement

READ (u,f) list

requests one logical record from the device associated with slot u in the device assignment table. The statement descriptions in this section use u to identify a specific I/O unit, f as the statement number of the FORMAT statement describing the type of data conversion, and list as a list of arguments to be input or output.

5.1 General I/O Statements

These statements cause the transfer of data between the computer and I/O devices.

5.1.1 Input/Output Argument Lists

An I/O statement which calls for the transmission of information includes a list of quantities to be transmitted. In an output statement this list consists of the variables to which the incoming data is to be assigned; in an output statement the list consists of the variables whose values are to be transmitted to the given I/O device. The list is ordered, and the order must be that in which the data words exist (input) or are to exist (output) in the I/O device. Any number of items in the list are transmitted. The remaining data is ignored. Conversely, if the items in the list exceed the data to be transmitted, succeeding superfluous records are transmitted until all items specified in the list have been transmitted.

A simple list has the form

C1, C2, ..., Cn

where each Ci is a variable, a subscripted variable, or an array identifier. Constants are not allowed as list items. The list reads from left to right. When an array identifier appears in the list, the entire array is to be transmitted before the next item in the list. Examples of Simple Lists:

```
Y, Y, Z
A, B(3), C, D (I+1, 4)
```

Indexing similar to that of the DO statement may be used to control the number of times a group of simple lists is to be repeated. The list elements thus controlled, and the index control itself, are enclosed in parentheses, and the contents of the parentheses are regarded as a single item of the I/O list. Examples are:

```
W, X(3), (Y (1), Z (I, K), I = 1, 10)
( (A (I, J) , I = 1, M), J = 1, N)
```

5.1.2 READ Statement

The READ statement is used to transfer data from any input device to the computer. The general READ statement can be used to read either BCD or binary information. The form of

the statement determines what kind of input will be performed.

The formatted READ statements have the general form

READ (u,f) list

or

READ (u,f)

Execution of this statement causes input from device u to be converted as specified by format statement f, the resulting values to be assigned to the items specified by list, if any. Format statements are discussed later in this section.

An unformatted READ statement has the general form

READ (u) list

or

READ (u)

Execution of this statement causes input from device u, in binary format, to be assigned to the items specified by list. If no list is given, one record will be read, but ignored. If the record contains more information words than the list requires, that part of the record is lost. If more elements are in the list than are in one record, an error condition will result. Examples of READ are:

```
READ (3, 13) A, B, C
READ (2, 10) A, (B (I), I = 1,5)
READ (1,3)
READ (5) I, J, K
READ (8)
```

5.1.3 WRITE Statement

The WRITE statement is used to transmit information from the computer to any I/O device. The WRITE statement closely parallels the READ statement in both format and operation.

The formatted WRITE statement has the general form

WRITE (u,f) list

or

WRITE (u,f)

Execution of this statement causes the list elements, if any, to be converted according to format statement f, and output into device u.

The formatted WRITE statement has the general form

WRITE (u) list

Execution of this statement causes output onto device u, in binary format, of all words specified by the list. Examples of WRITE are:

```
WRITE (1, 10) A, (B (1), (C (I, J), J=2, 10, 2), I=1,5)
WRITE (2,7) A,B,C
WRITE (5) W, X (3), Y (I * 1, 4), Z
```

5.2 Format Statements

These statements are used in conjunction with the general I/O statements. They specify the type of conversion which is to be performed between the internal machine language and the external notation. FORMAT statements are not executed. Their function is to supply information to the object program.

5.2.1 Specifying FORMAT

The general form of the FORMAT statement is

FORMAT (S1, S2, ..., Sn)

where S1 .... Sn are data field descriptions. Breaking this format down further, the basic data field descriptor is written in the form

nkw.d

where n is a positive unsigned integer indicating the number of successive fields for which the data conversion will be performed according to the same specification. This is also known as the repeat count. If n is equal to 1, it may be omitted. The control character k indicates which type of conversion will be performed. This character may be I,E,F,D,P,L,A,H,X or O. The nonzero integer constant w specifies the width of the field. The integer constant d indicates the number of digits to the right of the decimal point.

Six of the nine control characters listed above provide for data conversion between internal machine language and external notation.

Internal	Type	External
Integer variable	I	Decimal integer
Real variable	E	Floating-point, scaled
Real variable	F	Floating-point
Double-precision variable	D	Floating-point, scaled
Logical variable	L	Letter T or F
Alphanumeric	A	Alphameric (BCD) characters
Octal	O	Alphameric representation of Octal digits

The other three control types are special purpose control characters:

Type	Purpose
P	Used to set a scale factor for use with E,F, and D conversions.
X	Provides for skipping characters in input or specifying blank characters in output.
H	Designates Hollerith fields

FORMAT statements are not executed and therefore may be placed anywhere in the source program. Because they are referenced by READ or WRITE statements, each FORMAT statement must be given a statement number.

Commas (,) and slashes (/) are used as field separators. The comma is used to separate field descriptors, with the exception that a comma need not follow a field specified by an H or X control character. The slash is used to specify the termination of formatted records. A series of slashes is also a field separator. Multiple slashes are the equivalent of blank records between output records, or records skipped for input records. If the series of n slashes occurs at the beginning or the end of the FORMAT specifications, the number of input records

skipped or blank lines inserted in output is n. If the series of n slashes occurs in the middle of the FORMAT specifications, this number is n-1. An integer value cannot precede a slash.

For all field descriptors (with the exception of H and X), the field width must be specified. For those descriptors of the w.d type (described below), the d must be specified even if it is zero. The field width should be large enough to provide for all characters needed to separate it from other data values. Since the data value within a field is right justified, if the field specified is too small, the most significant characters of the value will be lost.

Successive items in the I/O list are transmitted according to successive descriptors in the FORMAT statement, until the entire I/O is satisfied. If the list contains more items than descriptors in the FORMAT statement, a new record must be begun. Control is transferred to the preceding left parenthesis where the same specifications are used again until the list is complete.

Field descriptors (except H and X) are repeated by preceding the descriptor with an unsigned integer constant (the repeat count). A group repeat count is used to enable the repetition of a group of field descriptors or field separators enclosed in parentheses. The group count is placed to the left of the parenthesis. Two levels of parentheses (not including those enclosing the FORMAT specification) are permitted.

The field descriptors in the FORMAT must be the same type as the corresponding item in the I/O list, i.e., integer quantities require integer (I) conversion; real quantities require real (E or F) conversion, etc. Example:

```
FORMAT (17, F10.3)
FORMAT (13, 17/E10.4, E10.4)
FORMAT (214, 3(15, D10.3))
```

5.2.2 Conversion of Numeric Data

5 .2.2.1 I-Type Conversion - (Field Descriptor Iw or nIw)

The number of characters specified by w is converted as a decimal integer. On input, the number in the input field by w is converted to a binary integer. A minus sign indicates a negative number. A plus sign, indicating a positive number, is optional. The decimal point is illegal. If there are blanks, they must precede the sign or first digit. Al embedded blanks are interpreted as zero digits.

On output, the converted number is right justified. If the number is smaller than the field w allows, the leftmost spaces are filled with blanks. If an integer is too large, the most

significant digits are truncated and lost. Negative numbers have a minus sign just preceding their most significant digit if sufficient spaces have been reserved. No sign indicates a positive number. Examples (b indicates blank):

Format Descriptor	Input	Internal	Output
15	bbbbb	+00000	bbbb0
13	-b5	-5	b-5
18	bbb12345	+12345	bbb12345

5.2.2.2 E-Typs Conversion - (Field Descriptor Ew.d or nEw.d)

The number of characters specified by w is converted to a floating-point number with d spaces reserved for the digits to the right of the decimal point. The w includes field d, spaces for a sign, the decimal point, plus four spaces for the exponent (written E ±XX) in addition to space for optional sign and one digit preceding the decimal point.

The input format of an E-type number consists of an optional sign, followed by a string of digits containing an optional decimal point, followed by an optional exponent. The exponent may have the character D or E. Input data can be any number of digits in length, although it must fall within the range of 0 to $\pm 10^{\pm 39}$. E output consists of a minus sign if negative (blank if positive), the digit 0, a decimal point, a string of digits rounded to d significant digits, followed by an exponent of the form E±XX. Examples are:

Format Descriptor	Input	Internal	Output
E10.4	00.2134E03	213.4	0.2134E+03
E9.4	0.2134E02	21.34	.2134E+02
E10.3	bb-23.0321	-23.0321	-0.230E+02

5.2.2.3 F-Type Conversion - (Field Descriptor Fw.d or nFw.d)

The number of characters specified by w is converted as a floating-point mixed number with d spaces reserved for the digits to the right of the decimal point. Input for F-type conversion is the same as that for E-type conversion, described above. The output consists of a minus sign if the number is negative (blank if positive), the integer portion of the number, a decimal point, and the fractional part of the number rounded to d significant digits.

Examples are:

Format Descriptor	Input	Internal	Output
F6.3	b13457	13.457	13.457
F6.3	313457	313.457	13.457
F9.2	-21367.	-21367.	-21367.00
F7.2	-21367.	-21367.	1367.00

5.2.2.4 D-Type Conversion - (Field Descriptor Dw.d or nDw.d)

The number of characters specified by w is converted as a double-precision floating-point number with the number of digits specified by d to the right of the decimal point. The input and output are the same as those for E-type conversion except that a D is used in place of the E in the exponent on output. Examples are:

Format Descriptor	Input	Internal	Output
D12.6	bb+21345E-03	21.345	0.213450D+02
D12.6	b 3456789012	3456.789012	0.345678D+04
D12.6	-12345.6D-02	- 123.456	0.123445D+03

5.2.3 P-Scale Factor (Field Descriptor nP or -nP)

This scale factor n is an integer constant. The scale factor has effect only on E-, F-, and D-type conversions. Initially, a scale factor of zero is implied. Once a P field descriptor has been processed, the scale factor established by n remains in effect for all subsequent E, F, and D descriptors within the same FORMAT statement until another scale factor is encountered.

For F, E, and D input conversions (when no exponent exists in the external field) the scale and D output, the fractional part is multiplied by 10n and the exponent is reduced by n.

Examples:

Format Descriptor	Input	Scale Factor	Internal	Output
F6.3	123456	-3	+123456.	23.456
E12.4	123456	-3	+12345.6	bb0.0001E+08
D10.4	12.3456	+1	+1.23456	1.2345D+00

5.2.4 Conversion of Alphanumeric Data

5.2.4.1 A-Type Conversion (Field Descriptor Aw or nAw)

The number of alphanumeric characters specified by w is transmitted according to list specifi - cations. If the field width specified for input is greater than or equal to the data item cha- count (2 for integer or logical, 4 for real or 8 for double precision real), the rightmost charac- ters are stored internally. If the field width is less than the character count, the data is stored left-adjusted and the data item is filled with trailing blanks. For output, if the field width is greater than the item character count, the output is right adjusted and filled with leading blanks. If the output field width is less than the item character count, the leftmost characters are output.

5.2.4.2 H-Field Descriptor (Field Descriptor nH...)

The number of characters specified by n immediately following the H descriptor are trans- mitted to or from the external device. Blanks may be included in the alphanumeric string. The value of n must be greater than 0. On Hollerith input, n characters read from the ex- ternal device replace the n characters following the letter H in the format statement itself. In output mode, the n characters following the letter H, including blanks, are output. Examples are:

```
3HABC
17H THIS IS AN ERROR
```

An alternate means of writing the H descriptor in PDP-X fortran is by means of quotes. Thus the following cause identical effect:

```
10HABCDEFGHIJ
'ABCDEFGHIJ'
```

5.2.5 Logical Fields, L Conversion (Field Descriptor Lw or nLw)

The external format of a logical quantity is T or F. On L input, the first nonblank character must be a T or F. Leading blanks are ignored. For L output, if the internal value is false, an F is output. Otherwise a T is output. The F or T is preceded by w-1 leading blanks

5.2.6 Blank Fields, X Conversion (Field Descriptor nX)

The value of n is an integer number greater than 0. On X input, n characters are read but ignored. On X output, n spaces are output.

5.3 FORTRAN Statements Read in at Object Time

FORTRAN provides the facility of including the formatting data along with the input data. This is done by using an array name in place of the reference to a FORMAT statement label in any of the formatted I/O statements. For an array to be referenced in such a manner, the name of the variable FORMAT specification must appear in a DIMENSION statement, even if the size of the array is 1.

The statements have the general form:

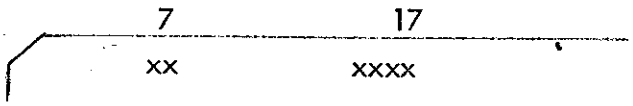
```
READ (u, name)
READ (u, name) list

WRITE (u, name)
WRITE (u, name) list
```

The form of the FORMAT specification which is to be inserted in the array is the same as the source program FORMAT statement, except that the word FORMAT is omitted. The FORMAT specification may be inserted in the array by using a data initialization statement, or by using a READ statement together with an A format For example, this facility can be used to specify at object time the format of a deck of cards to be read. The first card of the deck would contain the format statement,

```
1         10
(I7 , F10.3)
```

the subsequent cards would contain data in the general form,




```
      DIMENSION AA (10)
13    FORMAT (10A5)
      READ (3, 13) (AA(1), I = 1, 10)
        .
        .
      READ (3, AA) JJ, BOB
```

With the card reader assigned to device number 3, the first READ places the format statement from the first card into the array AA, and the second READ statement causes data from the subseauent cards to be read into JJ and BOB with format specifications 17 and F10.3 respectively.

5.4 Printing of a Formatted Record

When formatted records are prepared for printing, the first character of the record is not printed. It is used instead to determine vertical spacing as follows:

Character	Vertical Spacing Before Printing
Blank	One line
0	Two lines
1	Skip to first line of next page
+	No advance

Output of formatted records to other devices considers the first character as an ordinary character in the record.

5.5 Auxiliary I/O Statements

These statements manipulate the I/O file oriented devices. The u is an unsigned integer constant or integer variable specifying the device.

5.5.1 BACKSPACE Statement

The BACKSPACE statement has the general form

BACKSPACE u

Execution of this statement causes the I/O device identified by u to be positioned so that the record which had been the preceding record becomes the next record. If the unit u is positioned at its initial point, execution of this statement has no effect.

5.5.2 REWIND Statement

The REWIND statement has the general form

REWIND u

Execution of this statement causes the I/O device identified by u to be positioned at its initial point.

5.5.3 ENDFILE Statement

The ENDFILE statement has the general form

ENDFILE u

Execution of this statement causes an endfile record to be written on the I/O device identified by u.

5.5.4 Testing for End of File

Although this feature is not technically a part of the fortran compiler, it would seem reasonable to mention here that end of file can be sensed by means of the library logical function ENDFILE. Thus

```
IF (ENDFIL (1) ) GO TO 1ØØ
GO TO 2ØØ
```

would cause control to go to statement 100 if unit 1 was at end of file and to statement 2ØØ otherwise.

6. Specification Statements

Specification statements are nonexecutable because they do not generate instructions in the object program. They provide the compiler with information about the nature of the constants

and variables used in the program. They also supply the information required to allocate locations in storage for certain variables and/or arrays. All SPECIFICATION statements, with the exception of the DATA statement, must appear before any executable code generating statement. They must appear in this order: type statements, DIMENSION statements, COMMON statements, and EQUIVALENCE statements. EXTERNAL statements may appear anywhere after all type statements and before the executable code generating statements. The DATA statements may appear anywhere in the source program.

6.1 Type Statements

The type statements are of the forms

```
INTEGER a, b, c
REAL a, b, c
DOUBLE PRECISION a, b, c
LOGICAL a, b, c
```

where a, b, and c are variable names which may be dimensional or function names. A type statement is used to inform the compiler that the identifiers listed are variables or functions of a specified type, i.e., INTEGER, REAL, etc. It overrides any implicit typing, i.e., identifies which begin with the letters I, J, K, L, M, or N are implicitly of the INTEGER mode, those beginning with any other letter are implicitly of the REAL mode. The type statement may be used to supply dimension information. The variable or function names in each type statement are defined to be of that specific type throughout the program, the type may not change. Examples:

```
INTEGER ABC, IJK, XYZ
REAL A (2,4), I, J, K
DOUBLE PRECISION ITEM , GROUP
LOGICAL
```

6.2 Dimension Statement

The DIMENSION statement is used to declare arrays and to provide the necessary information to allocate storage for them in the object program. The general form is:

```
DIMENSION V (i1), V2 (i2), ... Vn (in)
```

where each V is the name of an array and each i is composed of one, two, or three unsigned integer constants separated by commas. The number of constants represents the number of dimensions the array contains; the value of each constant represents the maximum size of

each dimension. If the dimension information for the variable is given in a type statement or a COMMON statement, it must not be included in a DIMENSION statement. Example:

```
DIMENSION ITEM (150), ARRAY (50, 50)
```

When arrays are passed to subprograms, they must be redeclared in the subprogram. The mode, number of dimensions, and size of each dimension must conform to that declared by the calling program.

6.3 Common Statement

The COMMON statement provides a means of sharing memory storage between a program and its subprograms. The general form of the COMMON statement is:

```
COMMON /x1/a1/x2/a2/ .../xn/an
```

where each x is a variable which is a COMMON block name, or it can be blank. If x1 is blank, the first two slashes are optional. Each a represents a list of variables and arrays separated by commas. The list of elements pertaining to a block name ends with a new block name, with a blank COMMON block designation (two slashes), or the end of the statement.

The elements of a COMMON block, which are listed following the COMMON block name (or the blank name), are located sequentially in order of their appearance in the COMMON statement. An entire array is assigned in sequence. Block names may be used more than once in a COMMON statement, or may be used in more than one COMMON statement within the program. The entries so assigned are strung together in the given COMMON block in order of their appearance. Labeled COMMON blocks with the same name appearing in several programs or subprograms executed together must contain the same number of total words. The elements within the blocks, however, need not agree in name, mode, or oder. A blank COMMON may be any length. Examples:

```
COMMON A, B, C/XX/X, Y, Z
COMMON /A/X(3, 3), Y(2, 5)/ /Z(5, 10, 15)
```

The COMMON statement is a means of transferring data between programs. If one program contains the statements,

```
COMMON / N /AA, BB, CC
AA=3
BB=4
CC=5
```

and another program which is called later contains the statement,

COMMON /N/XX, YY, ZZ

then the latter program will find the values 3, 4, and 5 in its variables XX, YY, and ZZ, respectively, since variables in the same relative positions in COMMON statements share the same registers in memory.

6.4 Equivalence Statement

The EQUIVALENCE statement is used to permit two or more entities of the same size and type to share the same storage location. The general format of the EQUIVALENCE statement is:

EQUIVALENCE (k1), (k2), ..., (kn)

where each k represents a list of two or more variables or subscripted variables separated by commas. Each element in the list is assigned the same memory storage location. When two variables or array elements share the same storage location because of the use of an EQUIVALENCE statement, they may not both appear in COMMON statements within the same program. An example is:

EQUIVALENCE (A, B), (C(10), D(10), E(15))

6.5 External Statement

An EXTERNAL statement is used to pass a subprogram name on to another subprogram. The general form of an EXTERNAL statement is:

EXTERNAL y, z, ...

where y, z, ... are subroutine or function names.

6.6 Data Statement

The DATA statement is used to set variables or array elements to initial values at the time the object program is loaded. The general form of the DATA initialization statement is:

DATA k1/d1/, k2/d2/, ... kn/dn/

where each k is a list of variables or array elements (with constant subscripts) separated by commas, and each d is a corresponding list of constants with optional signs. The k list may not contain dummy arguments. There must be a one-to-one correspondence between the name list and the data list, except where the data list consists of a sequence of identical

constants. In such a case, the constant need be written only once, preceded b an integer constant indicating the number of repeats and an asterisk. A Hollerith constant may appear in the data list.

Variable or array elements appearing in a DATA statement may not be in blank COMMON. They may be in a labeled COMMON block and initially defined only in a BLOCK DATA subprogram. Some examples are:

```
DATA A, B, C/3*2.0/
DATA X(1), X(2), X(3), X(4)/0.0, 0.1, 0.2, 0.3
```

7. Subprograms

There are five categories of subprograms:

a. Statement Functions
b. Intrinsic or Library Functions
c. External Functions
d. External Subroutines
e. Block Data Subprograms

The first three categories of subprograms are referred to as functions. Functions and subroutines differ in the following two respects. The former are called by writing the name of the function and an argument list in a standard arithmetic expression; the latter by using a CALL statement. Block data is a special purpose subprogram used for data initialization purposes.

7.1 Statement Functions

A statement function is defined by a single statement similar in form to that of an arithmetic assignment statement. If is defined internally to the program unit by which it is referenced. Statement functions must follow all specification statements and precede any executable statements of the program unit of which they are a part. The general format of a statement function is:

f (a1, a2, ..., an) = e

where f is a function name; the a's are nonsubscripted variables, known as dummy arguments, which are to be used in evaluating the function; and e is an expression.

The value of a function is a real quantity unless the name of the function begins with I, J, K, L, M, or N; in which case it is an integer quantity, or the function type may be defined by using the appropriate specification statement. Since the arguments are dummy variables, their names are unimportant, except to indicate mode, and may be used else-

where in the program, including within the expression on the right side of the statement function.

The expression of a statement function, in addition to containing nonsubscripted dummy arguments, may only contain:

a. Non-Hollerith constants
b. Variable references
c. Intrinsic function references
d. References to previously defined statement functions
3. External function references

A statement function is called any time the name of the function appears in any FORTRAN arithmetic expression. The actual arguments must agree in order, number, and type with the corresponding dummy arguments. Execution of the statement function reference results in the computations indicated by the function definition. The resulting quantity is used in the expression which contains the function reference. Examples are:

```
A(X) = 3.2+SQRT (5.7* X**2)
SUM (A,B,C) = A+B+C
FUNC (A,B) = 3.*A/B**2.+Z
```

7.2 Intrinsic or Library Functions

Intrinsic or library functions are predefined subprograms that are a part of the FORTRAN system library. The type of each intrinsic function and its arguments are predefined and cannot be changed. An intrinsic function is referenced by using its function name with the appropriate arguments in an arithmetic statement. The arguments may be arithmetic expressions, subscripted or simple variables, constatns, or other intrinsic functions. Table 1 describes the PDP-X intrinsic functions.

TABLE 1 INTRINSIC FUNCTIONS

Intrinsic Functions	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute value	$\|a\|$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	a_1 (mod a_2)	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max (a_1, a_2, ...)	2	AMAXO	Integer	Real
			AMAXI	Real	Real
			MAXO	Integer	Integer
			MAXI	Real	Integer
			DMAXI	Double	Double
Choosing smallest value	Min (a_1, a_2, ...)	2	AMINO	Integer	Real
			AMINI	Real	Real
			MINO	Integer	Integer
			MINI	Real	Integer
			DMINI	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	Sign of a_2 times $\|a_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
			DSIGN	Double	Double
Positive difference	a_1 - Min (a_1, a_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain most significant part of double precision argument		1	SNGL	Double	Real
Express single precision argument in double precision form		1	DBLE	Real	Double

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]\, a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

7.3 External Functions

An external function is an independently written program which is executed whenever its name appears in another program. The general form in which an external function is written is:

```
+FUNCTION NAME (a1, a2, ..., an)

(FORTRAN statements)
   :
   :
NAME = final calculation
RETURN
END
```

where t is either INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or is a blank, NAME is the symbolic name of the function to be defined; and the a's are dummy arguments which are nonsubscripted variable names, array names, or other external function names.

The first letter of the function name implicitly determines the type of function. If that letter is I, J, K, L, M, or N, the value of the function is INTEGER. If it is any other letter, the value is REAL. This can be overridden by preceding the word FUNCTION with the specific type name. The symbolic name of a function is one to six alphanumeric characters, the first of which must be the alphabetic name and must not appear in any nonexecutable statement of the function subprogram except in the FUNCTION statement where it is named. The function name must also appear at least once as a variable name within the subprogram. During every execution of the subprogram, the variable must be defined before leaving the function subprogram. Once defined, it may be referenced or redefined. The value of this variable at the time any RETURN statement in the subprogram is encountered is called the value of the function. There must be at least one argument in the FUNCTION statement. If a dummy argument is an array name, an appropriate DIMENSION statement is necessary. The dummy argument names may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram. The function subprogram may contain any FORTRAN statements with the exception of a BLOCK DATA, SUBROUTINE, or another FUNCTION statement. It, of course, cannot contain any statement which references itself, either directly or indirectly.

A function subroutine must contain at least one RETURN statement. The general form is:

```
RETURN
```

This signifies the logical end of the subprogram and returns control and the computed value to the calling program.

An END statement, described in section 4.11, signals the compiler that the physical end of the subprogram has been reached.

An external function is called by using its function name, followed by an actual argument list enclosed in parentheses, in an arithmetic or logical expression. The actual arguments must correspond in number, order, and type to the dummy arguments. An actual argument may be one of the following:

a. A variable name
b. An array element name
c. An array name
d. Any other expression
e. The name of an external function or subroutine

Table 2 contains the basic external functions supplied by the FORTRAN system.

TABLE 2 EXTERNAL FUNCTIONS

Basic External Function	Definition	No. of Arguments	Symbolic Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
Natural logarithm	$\log_e (a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
Common logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
		2	DLOG10	Double	Double
Trigonometric sine	sin (a)	1	SIN	Real	Real
		1	DSIN	Double	Double
Trigonometric cosine	cos (a)	1	COS	Real	Real
		1	DCOS	Double	Double
Hyperbolic tangent	tanh (a)	1	TANH	Real	Real
Square root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
Arctangent	arctan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	a_1 (mod a_2)	2	DMOD	Double	Double

*The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]\, a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

7.4 Subroutines

A subroutine is defined externally to the program unit which references it. It is similar to an external function in that both contain the same sort of dummy arguments, and both require an END statement. With a subroutine, however, all communication with the calling program is via the argument list. The general form of a subroutine is:

SUBROUTINE NAME (a1, a2, ..., an)

or

SUBROUTINE NAME

where NAME is the symbolic name of the subroutine subprogram to be defined; and the a's are dummy arguments (there need not be any) which are nonsubscripted variable names, array names, or the dummy name of another subroutine or external function.

The name of a subroutine consists of one to six alphanumeric characters, the first of which is alphabetic. The symbolic names of the subroutines cannot appear in any statement of the subroutine except the SUBROUTINE statement itself. The dummy variables represent input and output variables. Any arguments used as output variables must appear on the left side of an arithmetic statement or an input list within the subprogram. If an argument is the name of an array, it must appear in a DIMENSION statement with the subroutine. The dummy argument names may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

The subroutine subprogram may contain any FORTRAN subprograms with the exception of FUNCTION, BLOCK DATA, or another SUBROUTINE statement. The logical termination of a subroutine is a RETURN statement. The physical end of the subroutine is an END statement.

A subroutine is referenced by a CALL statement, which has the general form

CALL NAME (a1, a2, ..., an)

or

CALL NAME

where NAME is the symbol name of the subroutine subprogram being referenced, and the a's are the actual arguments that are being supplied to the subroutine. The actual arguments in the CALL statement must agree in number, order, and type with the corresponding arguments in the SUBROUTINE subprogram. The array sizes must be the same. An actual

argument in the CALL statement may be one of the following:

a. A Hollerith constant
b. A variable name
c. An array element name
d. An array
e. Any other expression
f. The name of an external function or subroutine

7.5 Block Data Subprogram

The BLOCK DATA subprogram is a special subprogram used to enter data into a COMMON block during compilation. A BLOCK DATA statement takes the form

```
BLOCK DATA
```

This special subprogram contains only DATA, COMMON, EQUIVALENCE, DIMENSION, and TYPE statements. It cannot contain any executable statements. It can be used to initialize data only in a labeled COMMON block area; not in a blank COMMON block area. All elements of a given COMMON block must be listed in the COMMON statement, even if they do not all appear in a DATA statement. Data may be entered in more than one COMMON block in a single BLOCK DATA subprogram.

An END statement signifies the termination of a BLOCK DATA subprogram. An example is:

```
      BLOCK DATA
      DIMENSION X(4), Y(4)
      COMMON / NAME /A, B, C, I, J, X, Y
      DATA A, B, C/3*2.0/
      DATA X(1), X(2), X(3), X(4)/0.0,0.1,0.2,0.3/Y(1), Y(2),
     2      Y(3), Y(4)/1.oE-2, 1.0Er, 1.0E-4/
      END
```

Appendix - Differences Between PDP-9 Fortran and ASA Fortran

1. The quotation mark character is allowed in PDP-9 fortran. It is used to enclose alphameric data constants. It can be used in format statements, arguments of CALL statements and DATA statements. This feature is not in the ASA standard.

2. ASA fortran specifies nineteen continuation cards, for a total of 20 cards maximum in a statement. PDP-9 fortran can have only 5 continuation cards or 6 cards in total.

3. PDP-X Fortran has no complex data.

4. General integer expressions may be used for subscripts in PDP-X fortran. The ASA standards impose certain restrictions.

5. The G field descriptor is not allowed in PDP-X FORMAT statements.

6. The implied DO feature is not legal in a PDP-X DATA statement.

7. The size of arrays in subprograms is not adjustable to the size specified by the following program.

8. The 3 limits in the DO specification may be general integer expressions in PDP-X fortran.

9. An additional format descriptor, O for Octal, is allowed in PDP-X FORMAT statements.