DECSYSTEM

FORTRAN
Reference Manual

Order No. DEC-20-LFRMA-A-D

20

digital

FORTRAN
Reference Manual

Order No. DEC-20-LFRMA-A-D

digital equipment corporation • maynard. massachusetts

First Printing, January 1976

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

CONTENTS

CONTENTS (Cont)

Page

TABLES

PREFACE

The DECsystem-20 FORTRAN Reference Manual describes the FORTRAN language as implemented for the DECsystem-20 FORTRAN Language Processing System (referred to as DECsystem-20 FORTRAN).

The language manual is intended for reference purposes only; tutorial type text has been minimized. The reader is expected to have some experience in writing FORTRAN programs and to be familiar with the standard FORTRAN language set and terminology as defined in the American National Standard FORTRAN, X3.9-1966.

The descriptions of the DECsystem-20 FORTRAN extensions and additions to the standard FORTRAN language set are printed in ***bold face italic type.***

Operating procedures and descriptions of the DECsystem-20 programming environment are included in the appendices.

DECsystem-20 FORTRAN extensions to the 1966
ANSI standard set are printed in *boldface italic type.*

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The DECsystem-20 FORTRAN language set is compatible with and encompasses the standard set described in "American National Standard FORTRAN, X3.9-1966" (referred to as the 1966 ANSI standard set). DECsystem-20 FORTRAN also provides many extensions and additions to the standard set which greatly enhance the usefulness of DECsystem-20 FORTRAN and increases its compatibility with FORTRAN language sets implemented by other major computer manufacturers. In this manual the DECsystem-20 FORTRAN extensions and additions to the 1966 ANSI standard set are printed in ***boldface italic type.***

A DECsystem-20 FORTRAN source program consists of a set of statements constructed using the language elements and the syntax described in this manual. A given FORTRAN statement will perform anyone of the following functions:

a. It will cause operations such as multiplication, division, and branching to be carried out.

b. It will specify the type and format of the data being processed.

c. It will specify the characteristics of the source program.

FORTRAN statements are comprised of key words (i.e., words which are recognized by the compiler) used with elements of the language set: constants, variables, and expressions. There are two basic types of FORTRAN statements: executable and nonexecutable.

Executable statements specify the action of the program; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and the kind of subprograms that may be included in the program. The compilation of executable statements results in the creation of executable code in the object program. Nonexecutable statements provide information only to the compiler, they do *not* create executable code.

In this manual the FORTRAN statements are grouped into twelve categories, each of which is described in a separate chapter. The name, definition, and chapter reference for each statement category are given in Table 1-1.

The basic FORTRAN language elements (i.e., constants, variables, and expressions), the character set from which they may be formed, and the rules which govern their construction and use are described in Chapters 2 through 4.

Table 1-1
FORTRAN Statement Categories

Category Name	Description	Chapter Reference
Compilation Control Statements	Statements in this category identify programs and indicate their end.	5
Specification Statements	Statements in this category declare the properties of variables, arrays, and functions.	6
DATA Statement	This statement assigns initial values to variables and array elements.	7
Assignment Statements	Statements in this category cause named variables and/or array elements to be replaced by specified (assigned) values.	8
Control Statements	Statements in this category determine the order of execution of the object program and terminate its execution.	9
Input/Output Statements	Statements in this category transfer data between internal storage and a specified input or output medium.	10
NAMELIST Statement	*This statement establishes lists that are used with certain input/output statements to transfer data which appears in a special type of record.*	11
File Control Statements	*Statements in this category identify, open and close files and establish parameters for input and output operations between files and the processor.*	12
FORMAT Statement	This statement is used with certain input/output statements to specify the form in which data appears in a FORTRAN record on a specified input/output medium.	13
Device Control Statements	Statements in this category enable the programmer to control the positioning of records or files on certain peripheral devices.	14
SUBPROGRAM Statements	Statements in this category enable the programmer to define functions and subroutines and their entry points.	15
BLOCK DATA Statements	Statements in this category are used to declare data specification subprograms which may initialize common storage areas.	16

CHAPTER 2
CHARACTERS AND LINES

2.1 CHARACTER SET

The digits, letters, and symbols recognized by DECsystem-20 FORTRAN are listed in Table 2-1. The remainder of the ASCII-1968 character set[1], although acceptable within literal constants or comment text, causes a fatal error in other contexts. An exception is CTRL/Z which, when used in terminal input, means end-of-file.

NOTE

Lower case alphabetic characters are treated as upper case outside the context of Hollerith constants, literal strings, and comments.

Table 2-1
DECsystem-20 FORTRAN Character Set

Letters		
A,a	J,j	S,s
B,b	K,k	T,t
C,c	L,l	U,u
D,d	M,m	V,v
E,e	N,n	W,w
F,f	O,o	X,x
G,g	P,p	Y,y
H,h	Q,q	Z,z
I,i	R,r	

(continued)

[1] The complete ASCII-1968 character set is defined in the X3.4-1968 version of the "American National Standard for Information Interchange," and is given in Appendix A.

Table 2-1 (Cont)
DECsystem-20 FORTRAN Character Set

Digits	
0	5
1	6
2	7
3	8
4	9

Symbols

! Exclamation Point	, Comma
" Quotation Marks	- Hyphen (Minus)
# Number Sign	. Period (Decimal Point)
$ Dollar Sign	/ Slant (slash)
& Ampersand	: Colon
' Apostrophe	; Semicolon
(Opening Parenthesis	< Less Than
) Closing Parenthesis	= Equals
* Asterisk	> Greater Than
+ Plus	^ Circumflex

Line Termination Characters

Line Feed
Form Feed
Vertical Tab

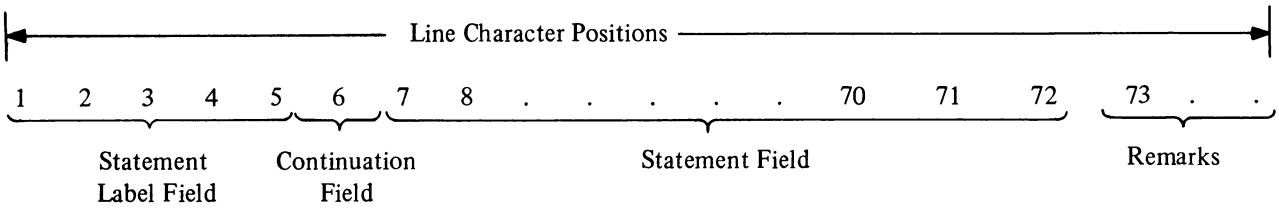
Line Formatting Characters

Carriage Return
Horizontal Tab
Blank

Note that horizontal tabs normally advance the character position pointer to the next position that is an even multiple of 8. An exception to this is the initial tab which is defined as a tab that includes or starts in character position 6. (Refer to Section 2.3.1 for a description of initial and continuation line types.) Tabs within literal specifications count as one character even though they may advance the character position pointer as many as eight places.

2.2 STATEMENT, DEFINITION, AND FORMAT

Source program statements are divided into physical lines. A line is defined as a string of adjacent character positions, terminated by the first occurrence of a line termination character regardless of context. Each line is divided into four fields:



2.2.1 Statement Label Field and Statement Numbers

A one to five digit number may be placed in the statement label field of an initial line to identify the statement. Any source program statement that is referenced by another statement must have a statement number. Statement numbers may be any number from 1 to 99999; leading zeroes and all blanks in the label field are ignored (e.g., the numbers 00105 and 105 are both accepted as statement number 105). The statement numbers given in a source program may be assigned in any order; however, each statement number must be unique with respect to all other statements in the program. Non executable statements, with the exception of FORMAT statements, cannot be labeled.

When source programs are entered into the system via a standard user terminal, an initial tab may be used to skip all or part of the label field.

If an initial tab is encountered during compilation, FORTRAN examines the character immediately following the tab to determine the type of line being entered. If the character following the tab is one of the digits 1 through 9, FORTRAN considers the line as a continuation line and the second character after the tab as the first character of the statement field. If the character following the tab is other than one of the digits 1 through 9, FORTRAN considers the line to be an initial line and the character following the tab is considered to be the first character of the statement field. The character following the initial tab is considered to be in character position 6 in a continuation line, and in character position 7 in an initial line.

2.2.2 Line Continuation Field

Any alphanumeric character (except a blank or a zero) placed in this field (position 6) identifies the line as a continuation line (see Paragraph 2.3.1 for description).

Whenever a tab is used to skip all or part of the label field of a continuation line, the next character entered must be one of the digits 1 through 9 to identify the line as a continuation line.

2.2.3 Statement Field

Any FORTRAN statement may appear in this field. Blanks (spaces) and tabs do not affect compilation of the statement and may be used freely in this field for appearance purposes, with the exception of textual data given within either a literal or Hollerith specification where blanks and tabs are significant characters.

2.2.4 Remarks

In lines comprised of 73 or more character positions, only the first 72 characters are interpreted by FORTRAN. (Note that tabs generally occupy more than one character position, advancing the counter to the next character position that is an even multiple of eight.) All other characters in the line (character positions 73, 74 . . . etc.) are treated as remarks and do not affect compilation.

Note that remarks may also be added to a line in character positions 7 through 72 provided the text of the remark is preceded by the symbol ! (refer to Paragraph 2.3.3).

2.3 LINE TYPES

A line in a DECsystem-20 FORTRAN source program can be

a. an initial line

b. a continuation line

c. *a multi-statement line*

d. a comment line

e. *a debug line*

f. a blank line.

Each of the foregoing line types is described in the following paragraphs.

2.3.1 Initial and Continuation Line Types

A FORTRAN statement may occupy the statement fields of up to 20 consecutive lines. The first line in a multi-line statement group is referred to as the "initial" line; the succeeding lines are referred to as continuation lines.

Initial lines may be assigned a statement number and must have either a blank or a zero in their continuation line field (i.e., character position 6).

If an initial line is entered via a keyboard input device, an initial tab may be used to skip all or part of the label field. An initial tab used for this purpose must be followed immediately by a nonnumeric character (i.e., the first character of the statement field must be nonnumeric).

Continuation lines cannot be assigned statement numbers: they are identified by any alphanumeric character (except for a blank or zero) placed in character position 6 of the line (i.e., continuation line field). The label field of a continuation line is treated as remark text.

If a continuation line is being entered via a keyboard, an initial tab may be used to skip all or part of the label field; however, the tab must be followed immediately by a numeric character other than zero. The tab-numeric combination identifies the line as a continuation line.

Note that blank lines, comments, and debug lines that are treated like comments, i.e., debug lines that are not compiled with the rest of the program (refer to section 2.3.4), terminate a continuation sequence.

Following is an example of a four line FORTRAN FORMAT statement using initial tabs:

```
105     FORMAT (1H1,17HINITIAL CHARGE = ,F10.6,10H    COULOME,6X,
     213HRESISTANCE = ,F9.3,6H    OHM/15H CAPACITANCE = ,F10.6,
     38H    FARAD,11X,13HINDUCTANCE = ,F7.3,8H    HENERY///
     421H     TIME       CURRENT/7H      MS,10X,2HMA///)
```

Continuation Line Characters (i.e., 2, 3, and 4)

2.3.2 Multi-Statement Lines

More than one FORTRAN statement may be written in the statement field of one line. The rules for structuring a multi-statement line are:

a. successive statements must be separated by a semicolon (;)

b. only the first statement in the series can have a statement number

c. statements following the first statement cannot be a continuation of the preceding statement

d. the last statement in a line may be continued to the next line if the line is made a continuation line.

An example of a multi-statement line is:

```
450      DIST=RATE * TIME ;TIME=TIME+0.05 ;CALL PRIME(TIME,DIST)
```

2.3.3 Comment Lines and Remarks

Lines that contain descriptive text only are referred to as comment lines. Comment lines are commonly used to identify and introduce a source program, to describe the purpose of a particular set of statements, and to introduce subprograms.

The rules for structuring a comment line are:

a. One of the characters C (or c), $,/,*, or ! must be in character position 1 of the line to identify it as a comment line.

b. The text may be written into character positions 2 through the end of the line.

c. Comment lines may appear anywhere in the source program, but may not precede a continuation line because comments terminate a continuation sequence.

d. A large comment may be written as a sequence of any number of lines. However, each line must carry the identifying character (C,$,/,*, or !) in its first character position.

The following is an example of a comment that occupies more than one line.

```
CSUBROUTINE - A12
CTHE PURPOSE OF THIS SUBROUTINE IS
CTO FORMAT AND STORE THE RESULTS OF
CTEST PROGRAM HEAT TEST-1101
```

Comment lines are printed on all listings but are otherwise ignored by the compiler.

A remark may be added to any statement field, in character positions 7 through 72, provided the symbol ! precedes the text. For example, in the line

```
IF(N.EQ.0)STOP! STOP IF CARD IS BLANK
```

the character group "Stop if card is blank" is identified as a remark by the preceding ! symbol. Remarks do not result in the generation of object program code, but they will appear on listings. The symbol !, indicating a remark, must appear outside the context of a literal specification.

Note that characters appearing in character positions 73 and beyond are automatically treated as remarks, so that the symbol ! need not be used (refer to Paragraph 2.2.4).

2.3.4 Debug Lines

As an aid in program debugging a D (or d) in character position 1 of any line causes the line to be interpreted as a comment line, i.e., not compiled with the rest of the program unless the / Include switch appears in the command string. (Refer to Appendix C for a description of the compile switch options.) When the / Include switch is present in the command string the D (or d) in character position 1 is treated as a blank so that the remainder of the line is compiled as an ordinary (noncomment) line. Note that the initial and all continuation lines of a debug statement must contain a D (or d) in character position 1.

2.3.5 Blank Lines

Lines consisting of only blanks, tabs, or no characters may be inserted anywhere in a FORTRAN source program except immediately preceding a continuation line, because blank lines are by definition initial lines and as such terminate a continuation sequence. Blank lines are used for formatting purposes only; they cause blank lines to appear in their corresponding positions in object program listings; otherwise, they are ignored by the compiler.

2.3.6 Line-Sequenced Input

FORTRAN optionally accepts line-sequenced files as produced by EDIT, the DECsystem-20 editor. These sequence numbers are used in place of the listing line numbers normally generated by FORTRAN.

2.4 ORDERING OF DECSYSTEM-20 FORTRAN STATEMENTS

The order in which FORTRAN Statements appear in a program unit is important. That is, certain types of statements have to be processed before others in order to guarantee that compilation takes place as expected. The proper sequence for FORTRAN statements is summarized by the following diagram.

<table>
<tr><td rowspan="6">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, Subprogram, or
BLOCK DATA Statements</td></tr>
<tr><td rowspan="5">FORMAT Statements</td><td colspan="2">IMPLICIT Statements</td></tr>
<tr><td colspan="2">PARAMETER Statements</td></tr>
<tr><td colspan="2">DIMENSION, COMMON,
EQUIVALENCE, EXTERNAL,
NAMELIST, or Type
Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement
Function
Definitions</td></tr>
<tr><td>Executable
Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Horizontal lines indicate the order in which FORTRAN statements must appear. That is, the statements in the horizontal sections cannot be interspersed. For example, all PARAMETER statements must appear after all IMPLICIT statements and before any DATA statements, i.e., PARAMETER, IMPLICIT, and DATA statements cannot be interspersed. Statement function definitions must appear after IMPLICIT statements and before executable statements.

Vertical lines indicate the way in which certain types of statements may be interspersed. For example, DATA statements may be interspersed with statement function definitions and executable statements. FORMAT statements may be interspersed with IMPLICIT statements, parameter statements, other specification statements, DATA statements, statement function definitions, and executable statements. The only restrictions on the placement of FORMAT statements are that they must appear after any PROGRAM, FUNCTION, SUBPROGRAM, and BLOCK DATA statements, and before the END statement.

Special Cases:

a. The placement of an INCLUDE statement is dictated by the types of statements to be INCLUDEd.

b. The ENTRY statement is allowed only in functions or subroutines. All executable references to any of the dummy parameters must physically follow the ENTRY statement unless the references appear in the function definition statement, the subroutine, or in a preceding ENTRY statement.

c. BLOCK DATA subprograms cannot contain any executable statements, statement functions, FORMAT statements, EXTERNAL statements, or NAMELIST statements. (Refer to section 16.1.)

FORTRAN expects users to adhere to the foregoing ordering guidelines and issues warning messages when statements are out of place.

DECsystem-20 FORTRAN extensions to the 1966
ANSI standard set are printed in *boldface italic type.*

CHAPTER 3
DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS

3.1 DATA TYPES

The data types permitted in DECsystem-20 FORTRAN source programs are

a. integer

b. real

c. double precision

d. complex

e. octal

f. double octal

g. literal

h. statement label, and

i. logical.

The use and format of each of the foregoing data types are discussed in the descriptions of the constant having the same data type (Paragraphs 3.2.1 through 3.2.8).

3.2 CONSTANTS

Constants are quantities that do not change value during the execution of the object program.

The constants permitted in DECsystem-20 FORTRAN are listed in Table 3-1.

Table 3-1
Constants

Category	Constant(s) Types
Numeric	Integer, real, double precision, complex, and ***octal***
Truth Values	Logical
Literal Data	Literal
Statement Label	***Address of FORTRAN statement label***

3.2.1 Integer Constants

An integer constant is a string of from one to eleven digits which represents a whole decimal number (i.e., a number without a fractional part). Integer constants must be within the range of $-2^{35}-1$ to $+2^{35}-1$ (i.e., -34359738367 to +34359738367). Positive integer constants may optionally be signed; negative integer constants must be signed. Decimal points, commas, or other symbols are not permitted on integer constants (except for a preceding sign, + or -). Examples of *valid* integer constants are:

```
 345
+345
-345
```

Examples of *invalid* integer constants are:

```
+345.    (use of decimal point)
 3,450   (use of comma)
 34.5    (use of decimal point; not a whole number)
```

3.2.2 Real Constants

A real constant may have any of the following forms:

a. A basic real constant: a string of decimal digits followed immediately by a decimal point which may optionally be followed by a fraction (e.g., 1557.00).

b. A basic real constant followed immediately by a decimal integer exponent written in E notation (i.e., exponential notation) form (e.g., 1559.E2).

c. An integer constant (no decimal point) followed by a decimal integer exponent written in E notation (e.g., 1559E2).

Real constants may be of any size; however, each will be rounded to fit the precision of 27 bits (i.e., 7 to 9 decimal digits).

Precision for real constants is maintained (approximately) to eight digits.[1]

[1] This is an approximation, the exact precision obtained will depend on the numbers involved.

The exponent field of a real constant written in E notation form cannot be empty (i.e., blank), it must be either a zero or an integer constant. The magnitude of the exponent must be greater than -38 and equal to or less than +38 (i.e., $-38 < n \leqslant 38$). The following are examples of *valid* real constants.

-98.765	
7.0E+0	(i.e., 7.)
.7E-3	(i.e., .0007)
5E+5	(i.e., 500000.)
50115.	
50.E1	(i.e., 500.)

The following are examples of *invalid* real constants.

72.6E75	(exponent is too large)
.375E	(exponent incorrectly written)
500	(no decimal point given)

3.2.3 Double Precision Constants

Constants of this type are similar to real constants written in E notation form; the direct differences between these two constants are:

a. Double precision constants depending on their magnitude have precision to 16 or 18 places, rather than the 8-digit precision obtained for real constants.

b. Each double precision constant occupies two storage locations.

c. The letter D, instead of E, is used in double precision constants to identify a decimal exponent.

Both the letter D and an exponent (even of zero) are required in writing a double precision constant. The exponent given need only be signed if it is negative; its magnitude must be greater than -38 and equal to or less than +38 (i.e., $-38 < n \leqslant +38$). The range of magnitude permitted a double precision constant depends on the type of processor present in the system on which the source program is to be compiled and run. The permitted range is 0.14×10^{-38} to $3.4 \times 10^{+38}$.

The following are *valid* examples of double precision constants.

7.9D03	(i.e., 7900)
7.9D+03	(i.e., 7900)
7.9D-3	(i.e., .0079)
79D03	(i.e., 79000)
79D0	(i.e., 79)

The following are *invalid* examples of double precision constants.

7.9D99	(exponent is too large)
7.9E5	(denotes a single precision constant)

3.2.4 Complex Constants

A complex constant can be represented by an ordered pair of integer, real or octal constants written within parentheses and separated by a comma. For example, (.70712, -.70712) and (8.763E3, 2.297) are complex constants.

In a complex constant the first (leftmost) real constant of the pair represents the real part of the number, the second real constant represents the imaginary part of the number. Both the real and imaginary parts of a complex constant can be signed.

The real constants that represent the real and imaginary parts of a complex constant occupy two consecutive storage locations in the object program.

3.2.5 Octal Constants

Octal numbers (radix 8) may be used as constants in arithmetic expressions, logical expressions, and data statements. Octal numbers up to 12 digits in length are considered standard octal constants; they are stored right-justified in one processor storage location. When necessary, standard octal constants are padded with leading zeroes to fill their storage location.

If more than 12 digits are specified in an octal number, it is considered a double octal constant. Double octal constants occupy two storage locations and may contain up to 24 right-justified octal digits; zeroes are added to fill any unused digits.

If a single octal constant is to be assigned to a double precision or complex variable, it is stored, right-justified, in the high order word of the variable. The low order portion of the variable is set to zero.

If a double octal constant is to be assigned to a double precision or complex variable, it is stored right-justified starting in the low order (rightmost) word and precedes leftwards into the high order word.

All octal constants must be

- ***a. preceded by a double quote (") to identify the digits as octal (e.g., "777), and***
- ***b. signed if negative but optionally signed if positive.***

The following are examples of valid octal constants:

```
"123456700007
"123456700007
+"12345
-"7777
"-7777
```

The following are examples of invalid octal constants:

```
"12368          (contains a radix digit)
7777            (no identifying double quotes)
```

When an octal constant is used as an operand in an expression, its form (i.e., bit pattern) is not converted to accommodate it to the type of any other operand. For example, the subexpression (A+"202 400 000 000) has as its result the sum of A with the floating point number 2.0; while the subexpression (I+"202 400 000 000) has as its result the sum of I with a large integer.

When a double octal constant is combined in an expression with either an integer or real variable, only the contents of the high order location (leftmost) are used.

3.2.6 Logical Constants

The Boolean values of truth and falsehood are represented in FORTRAN source programs as the logical constants .TRUE. and .FALSE.. Logical constants are always written enclosed by periods as in the preceding sentence.

Logical quantities may be operated on in arithmetic and logical statements. Only the sign bit of a numeric used in a logical IF statement is tested to determine if it is true (sign is negative) or false (sign is positive).

3.2.7 Literal Constants

A literal constant may be either of the following:

a. ***A string of alphanumeric and/or special characters contained within apostrophes (e.g., 'TEST#5').***

b. A Hollerith literal, which is written as a string of alphanumeric and/or special characters preceded by nH (e.g., nHstring). In the prefix nH, the letter n represents a number which specifies the exact number of characters (including blanks) that follow the letter H; the letter H identifies the literal as a Hollerith literal. The following are examples of Hollerith literals:

```
2HAB,
14HLOAD TEST #124,
6H#124-A
```

NOTE
A tab (→|) in a Hollerith literal is counted as one character (e.g., 3H→|AB).

Literal constants may be entered into DATA statements as a string of

a. up to ten 7-bit ASCII characters for complex or double precision type variables, and

b. up to five 7-bit ASCII characters for all other type variables.

The 7-bit ASCII characters which comprise a literal constant are stored left-justified (starting in the high order word of a 2-word precision or complex literal) with blanks placed in empty character positions. Literal constants that occupy more than one variable are stored in successive variables in the list. The following example illustrates how the string of characters

A LITERAL OF MANY CHARACTERS

is stored in a six-element array called A.

```
DIMENSION A(6)
DATA A /'A LITERAL OF MANY CHARACTERS'/
```

A(1) is set to 'A_LIT'
A(2) is set to 'ERAL_'
A(3) is set to 'OF_MA'
A(4) is set to 'NY_CH'
A(5) is set to 'ARACT'
A(6) is set to 'ERS__'

3.2.8 Statement Label Constants

Statement labels are numeric identifiers that represent program statement numbers.

Statement label constants are written as a string of from one to five decimal digits which are preceded by either a dollar sign ($) or an ampersand (&). For example, either $11992 or &11992 may be used as statement labels.

Statement label constants are used only in the argument list of CALL statements to define the statement to return to in a multiple RETURN statement. (Refer to Chapter 15.)

3.3 SYMBOLIC NAMES

Symbolic names may consist of any alphanumeric combination of from one to six characters. ***More than six characters may be given but FORTRAN ignores all but the first six.*** The ***first*** character of a symbolic name must be an alphabetic character.

The following are examples of legal symbolic names:

A12345
IAMBIC
ABLE

The following are examples of illegal symbolic names:

#AMBIC	(symbol used as first character)
1AB	(number used as first character)

Symbolic names are used to identify specific items of a FORTRAN source program; these items, together with an example of a symbolic name and text reference for each, are listed in Table 3-2.

Table 3-2
Use of Symbolic Names

Symbolic Names Can Identify	For Example	For a detailed description See Paragraph
1. A Variable	PI, CONST, LIMIT	3.4
2. An Array	TAX	3.5
3. An Array element	TAX (NAME,INCOME)	3.5.1
4. Functions	MYFUNC, VALFUN	15.2
5. Subroutines	CALCSB, SUB2, LOOKUP	15.5
6. External	SIN, ATAN, COSH	15.4
7. COMMON Block Names	DATAR, COMDAT	6.5

3.4 VARIABLES

A variable is a datum (i.e., storage location) that is identified by a symbolic name and is not an array or an array element. Variables specify values which are assigned to them by either arithmetic statements (Chapter 8), DATA statements (Chapter 7), or at run time via I/O references (Chapter 10). Before a variable is assigned a value, it is termed an undefined variable and should not be referenced except to assign a value to it.

If an undefined variable is referenced, an unknown value is obtained.

The value assigned a variable may be either a constant or the result of a calculation which is performed during the execution of the object program. For example, the statement IAB=5 assigns the constant 5 to the variable IAB; in the statement IAB=5+B, however, the value of IAB at a given time will depend on the value of variable B at the time the statement was last executed.

The type of a variable is the type of the contents of the datum which it identifies. Variables may be

a. integer

b. real

c. logical

d. double precision, or

e. complex.

The type of a variable may be declared using either implicit or explicit type declaration statements (Chapter 6). However, if type declaration statements are not used, the following convention is assumed by FORTRAN:

a. Variable names which begin with the letters I, J, K, L, M, or N are integer variables.

b. Variable names which begin with any letter *other than* I, J, K, L, M, or N are real variables.

Examples of determining the type of a variable according to the foregoing convention are given in the following table.

Variable	Beginning Letter	Assumed Data Type
ITEMP	I	Integer
OTEMP	O	Real
KA123	K	Integer
AABLE	A	Real

3.5 ARRAYS

An array is an ordered set of data identified by an array name. Array names are symbolic names and must conform to the rules given in Paragraph 3.3 for writing symbolic names.

Each datum within an array is called an array element. Like variables, array elements may be assigned values; before an array element is assigned a value it is considered to be undefined and should not be referenced until it has been assigned a value. If a reference is made to an undefined array element the value of the element will be unknown and unpredictable.

Each element of an array is named by using the array name together with a subscript that describes the position of the element within the array.

3.5.1 Array Element Subscripts

The subscript of an array element identifier is given, within parentheses, as either one subscript quantity or a set of subscript quantities delimited by commas. The parenthesized subscript is written immediately after the array name. The general form of an array element name is AN (S1, S2,...Sn), where AN is the array name and S1 through Sn represent n number of subscript quantities. Any number of subscript quantities may be used in an element name; however, the number used must always equal the number of dimensions (Paragraph 3.5.2) specified for the array.

A subscript can be any compound expression (Chapter 4), for example:

a. Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, (A+B,C*5,D/2) and (A**3, (B/4+C) *E,3) are valid subscripts.

b. Arithmetic expressions used in array subscripts may be of any type but noninteger expressions (including complex) are converted to integer when the subscript is evaluated.

*c. A subscript may contain function references (Chapter 14). For example: TABLE (SIN (A) *B, 2, 3) is a valid array element identifier.*

d. Subscripts may contain array element identifiers nested to any level as subscripts. For example, in the subscript (I(J(K(L))),A+B,C) the first subscript quantity given is a nested 3-level subscript.

The following are examples of *valid* array element subscripts:

a. IAB (1,5,3)

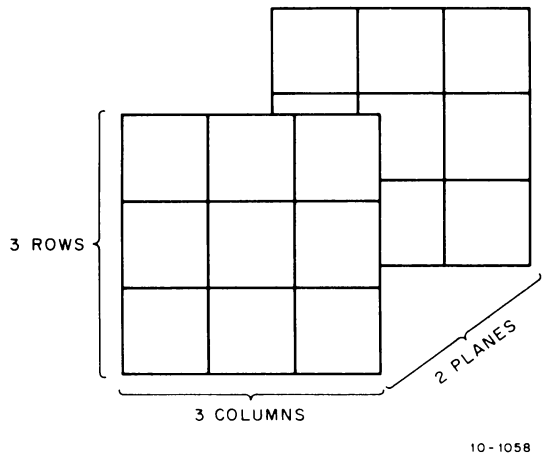
b. ABLE (A)

c. TABLE1 (10/C+K**2,A,B)

d. MAT(A,AB(2*L),.3*TAB(A,M+1,D),55)

3.5.2 Dimensioning Arrays

The size (i.e., number of elements) of an array must be declared in order to enable FORTRAN to reserve the needed amount of locations in which to store the array. Arrays are stored as a series of sequential storage locations. Arrays, however, are visualized and referenced as if they were single or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. For example, the following figure represents a 3-row, 3-column, 2-plane array.

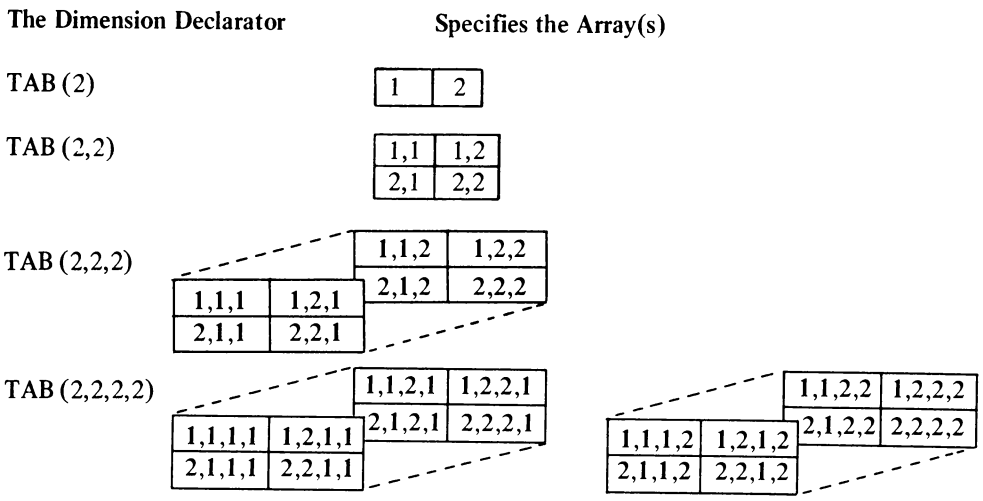


The size (i.e., number of elements) of an array is specified by an array declarator written as a subscripted array name. In an array declarator, however, each subscript quantity is a dimension of the array and must be either an integer, a variable, or an integer constant.

For example, TABLE (I,J,K) and MATRIX (10,7,3,4) are valid array declarators.

The total number of elements which comprise an array is the product of the dimension quantities given in its array declarator. For example, the array IAB dimensioned as IAB (2,3,4) has 24 elements (2 X 3 X 4 = 24).

Arrays are dimensioned only in the specification statements DIMENSION, COMMON, and type declaration (Chapter 6). Subscripted array names appearing in any of the foregoing statements are array declarators; subscripted array names appearing in any other statements are always array element identifiers. In array declarators the position of a given subscript quantity determines the particular dimension of the array (e.g., row, column, plane) which it represents. The first three subscript positions specify the number of rows, columns, and planes which comprise the named array; ***each following subscript given then specifies a set comprised*** of n-number (value of the subscript) of the previously defined sets. For example:



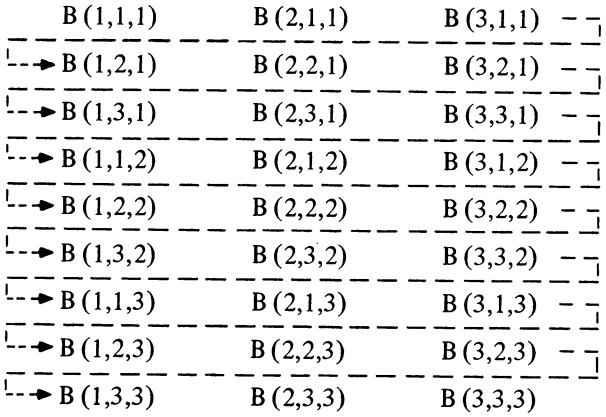
NOTE
DECsystem-20 FORTRAN permits any number of dimensions in an array declarator.

3.5.3 Order of Stored Array Elements

The elements of an array are arranged in storage in ascending order, with the value of the first subscript quantity varying between its maximum and minimum values most rapidly, and the value of the last given subscript quantity increasing to its maximum value least rapidly. For example, the elements of the array dimensioned as I(2,3) are stored in the following order:

I(1,1) → I(2,1) →I(1,2) → (2,2) → (1,3) → (2,3)

The following list describes the order in which the elements of the three-dimensional array (B(3,3,3)) are stored:



CHAPTER 4
EXPRESSIONS

4.1 ARITHMETIC EXPRESSIONS

Arithmetic expressions may be either simple or compound. Simple arithmetic expressions consist of an operand which may be

a. a constant

b. a variable

c. an array element

d. a function reference (see Chapter 14 for description), or

e. an arithmetic or logical expression written within parentheses.

Operands may be of type integer, real, double precision, complex, ***octal, or literal.***

The following are valid examples of simple arithmetic expressions:

105	(integer constant)
IAB	(integer variable)
TABLE (3, 4, 5)	(array element)
SIN (X)	(function reference)
(A+B)	(a parenthesized expression)

A compound arithmetic expression consists of two or more operands combined by arithmetic operators. The arithmetic operations permitted in FORTRAN and the operator recognized for each are given in Table 4-1.

Table 4-1
Arithmetic Operations and Operators

Operation	Operator	Example
1. Exponentiation	** or ↑	A**B or A↑B
2. Multiplication	*	A*B
3. Division	/	A/B
4. Addition	+	A+B
5. Subtraction	-	A-B

4.1.1 Rules for Writing Arithmetic Expressions

The following rules must be observed in structuring compound arithmetic expressions:

a. The operands comprising a compound arithmetic expression may be of different types. Table 4-2 illustrates all permitted combinations of data types and the type assigned to the result of each.

NOTE
Only one combination of data types, double precision with complex, is prohibited in DECsystem-20 FORTRAN.

b. An expression cannot contain two adjacent and unseparated operators. For example, the expression A*/B is not permitted.

c. All operators must be included, no operation is implied. For example, the expression A(B) does not specify multiplication although this is implied in standard algebraic notation. The expression A* (B) is required to obtain a multiplication of the elements.

d. In using exponentiation the base quantity and its exponent may be of different types. For example, the expression ABC**13 involves a real base and an integer exponent. The permitted base/exponent type combination and the type of the result of each combination is given in Table 4-3.

4.2 LOGICAL EXPRESSIONS

Logical expressions may be either simple or compound. Simple logical expressions consist of a logical operand which may be a logical type

a. constant

b. variable

c. array element

d. function reference (see Chapter 15), or

e. another expression written within parentheses.

Compound logical expressions consist of two or more operands combined by logical operators.

The logical operators permitted by DECsystem-20 FORTRAN and a description of the operation each provides are given in Table 4-4.

Table 4-2
Type of the Resultant Obtained From Mixed Mode Operations

Type of Argument 2

For operators +, -, *, /	Integer	Real	Double Precision	Complex	Logical	Octal	Double Octal	Literal
Integer 1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. None 4. None	1. Real 2. Real 3. From Integer to Real 4. None	1. Double Precision 2. Double Precision 3. From Integer to Double Precision 4. None	1. Complex 2. Complex 3. From Integer to Complex. Value used as Real part 4. None	1. Integer 2. Integer 3. None 4. None	1. Integer 2. Integer 3. None 4. None	1. Integer 2. Integer 3. None 4. High order word is used directly; low order word is ignored.	1. Integer 2. Integer 3. None 4. High order word is used directly; further words are ignored.
Real 1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Real 2. Real 3. None 4. From Integer to Real	1. Real 2. Real 3. None 4. None	1. Double Precision 2. Double Precision 3. Used directly as the high order word; low order word is zero. 4. None	1. Complex 2. Complex 3. Used directly as the Real part; imaginary part is zero. 4. None	1. Real 2. Real 3. None 4. None	1. Real 2. Real 3. None 4. None	1. Real 2. Real 3. None 4. High order word is used directly; low order word is ignored.	1. Real 2. Real 3. None 4. High order word is used directly; further words are ignored.

Table 4-5
Logical Operations, Truth Table

The result of the expression:	When P is:	and Q is:	Is:
.NOT.P	True	(Not	False
	False	Applicable)	True
P.AND.Q	True	True	True
	True	False	False
	False	True	False
	False	False	False
P.OR.Q	True	True	True
	True	False	True
	False	True	True
	False	False	False
P.XOR.Q	*True*	*True*	*False*
	True	*False*	*True*
	False	*True*	*True*
	False	*False*	*False*
P.EQV.Q	*True*	*True*	*True*
	True	*False*	*False*
	False	*True*	*False*
	False	*False*	*True*

Examples

Assume the following variables:

Variable	Type
REAL, RUN	Real
I,J,K	Integer
DP, D	Double Precision
L, A, B	Logical
CPX, C	Complex

Examples of valid logical expressions comprised of the foregoing variables are:

```
L.AND.B
(REAL*I) .XOR. (DP+K)
L.AND. A .OR. .NOT. (I-K)
```

Logical functions are performed bit-wise on the full 36-bit binary processor representation of the operands involved. The result of a logical operation is found by performing the specified function, simultaneously, for each of the corresponding bits in each operand. For example, consider the expression A=C.OR.D, where C= "456 and D= "201. The operation performed by the processor and the result is:

Word Bits	0	1 → 24	25	26	27	28	29	30	31	32	33	34	35
Operand C	0	0 → 0	0	0	1	0	0	1	0	1	1	1	0
Operand D	0	0 → 0	0	0	0	1	0	0	0	0	0	0	1
Result A	0	0 → 0	0	0	1	1	0	1	0	1	1	1	1

Table 4-6 is a truth table that illustrates all possible logical combinations of two one-bit binary operands (P and Q) and gives the result of each combination.

Table 4-6
Binary Logical Operations, Truth Table

The result of the expression:	When P is:	When And Q is:	Is:
.NOT.P	1	–	0
	0	–	1
P.AND.Q	1	1	1
	1	0	0
	0	1	0
	0	0	0
P.OR.Q	1	1	1
	1	0	1
	0	1	1
	0	0	0
P.XOR.Q	*1*	*1*	*0*
	1	*0*	*1*
	0	*1*	*1*
	0	*0*	*0*
P.EQV.Q	*1*	*1*	*1*
	1	*0*	*0*
	0	*1*	*0*
	0	*0*	*1*

4.2.1 Relational Expressions

Relational expressions are comprised of two expressions combined by a relational operator. The relational operator permits the programmer to test, quantitatively, the relationship between two arithmetic expressions.

The result of a relational expression is always a logically true or false value.

In FORTRAN, relational operators may be written either as a two-letter mnemonic enclosed within periods (e.g., .GT.) or *symbolically* using the *symbols* >, <, = *and* #. Table 4-7 lists both the mnemonic and symbolic forms of the FORTRAN relational operators and specifies the type of quantitative test performed by each operator.

Table 4-7
Relational Operators and Operations

Operators		Relation Tested
Mnemonic	*Symbolic*	
.GT.	>	Greater than
.GE.	>=	Greater than or equal to
.LT.	<	Less than
.LE.	<=	Less than or equal to
.EQ.	==	Equal to
.NE.	#	Not equal to

Relational expressions are written in the general form A_1 .OP. A_2, where A represents an arithmetic operand and .OP. is a relational operator.

Arithmetic operands of type integer, real, and double precision may be mixed in relational expressions.

Complex operands may be compared using only the operators .EQ (= =) and .NE. (#). Complex quantities are equal if the corresponding parts of both words are equal.

Examples

Assume the following variables:

Variables	Type
REAL, RON	Real
I, J, K	Integer
DP, D	Double Precision
L, A, B	Logical
CPX, C	Complex

Examples of *valid* relational expressions comprised of the foregoing variables are:

(REAL) .GT. 10
I == 5
C .EQ. CPX

Examples of *invalid* relational expressions comprised of the foregoing variables are:

(REAL) .GT 10 (closing period missing from operator)

C > CPX (complex operands can only be combined by .EQ. and .NE. operators)

Examples of *valid* expressions in which both logical and relational operators are used to combine the foregoing variables are:

```
(I .GT. 10) .AND. (J< =K)
((I*RON) = = (I/J)) .OR. K
(I .AND. K) # ((REAL) .OR. (RON))
C #CPX .OR. RON
```

4.3 EVALUATION OF EXPRESSIONS

The order of computation of a FORTRAN expression is determined by

a. the use of parentheses

b. an established hierarchy for the execution of arithmetic, relational, and logical operations and

c. the location of operators within an expression.

4.3.1 Parenthesized Subexpressions

In an expression all subexpressions written within parentheses are evaluated first. When parenthesized subexpressions are nested (one contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second and so on, until the value of the final parenthesized expression is computed. When more than one operator is contained by a parenthesized subexpression, the required computations are performed according to the hierarchy assigned operators by FORTRAN (Paragraph 4.3.2).

Example:

The separate computations performed in evaluating the expression

A+B/((A/B)+C)-C are:

a. A/B = *R1*

b. *R1*+C = *R2*

c. B/*R2* = *R3*

d. *R3*-C = *R4*

e. A+*R4* = *R5*

NOTE

R1 through R5 represent the interim and final results of the computations performed.

4.3.2 Hierarchy of Operators

The following hierarchy (i.e., order of execution) is assigned to the classes of FORTRAN operators:

first – arithmetic operators
second – relational operators
third – logical operators

The precedence assigned to the individual operators of the foregoing classes is specified (from highest to lowest) in Table 4-8.

With the exception of integer division and exponentiation, all operations on expressions or subexpressions involving operators of equal precedence are computed in any order that is algebraically correct.

A subexpression of a given expression may be computed in any order. For example, in the expression (F(X) + A*B) the function reference may be computed either before or after A*B.

Table 4-8
Hierarchy of FORTRAN Operators

Class	Level	Symbol or Mnemonic
ARITHMETIC	*First*	**
	Second	- (unary minus) and + (unary plus)
	Third	*,/
	Fourth	+,-
RELATIONAL	*Fifth*	.GT., .GE., .LT., .LE., .EQ., .NE.
	or	>, >=, <, <=, ==, #
LOGICAL	*Sixth*	.NOT.
	Seventh	.AND.
	Eighth	.OR.
	Ninth	.EQV., .XOR.

Operations specifying integer division are evaluated from left to right. For example, the expression I/J*K is evaluated as if it had been written as (I/J)*K.

When a series of exponentiation operations occurs in an expression, they are evaluated in order from right to left. For example, the expression A**2**B is evaluated in the following order:

*first 2**B = R1 (intermediate result)*
*second A**R1 = R2 (final result).*

4.3.3 Mixed Mode Expressions

Mixed mode expressions are evaluated on a subexpression by subexpression basis with the type of the results obtained converted and combined with other results or terms according to the conversion procedures described in Table 4-2.

Example

Assume the following:

Variable	Type
D	Double Precision
X	Real
I,J	Integer

The mixed mode expression D+X* (I/J) is evaluated in the following manner:

NOTE

R1, R2, and R3 represent the interim and final results of the computations performed.

a. (I/J) = R1 R1 is integer

b. X*R1 = R2 R1 is converted to type real and is multiplied by X to produce R2

c. D+R2 = R3 R2 is converted to type double precision and is added to D to produce R3

4.3.4 Use of Logical Operands in Mixed Mode Expressions

When logical operands are used in mixed mode expressions, the value of the logical operand is *not* converted in any way to accommodate it to the type of the other operands in the expression. For example, in L*R, where L is type logical and R is type real, the expression is evaluated without converting L to type real.

CHAPTER 5
COMPILATION CONTROL STATEMENTS

5.1 INTRODUCTION

Compilation control statements are used to identify DECsystem-20 FORTRAN programs and to specify their termination. Statements of this type do not affect either the operations performed by the object program or the manner in which the object program is executed. The three compilation control statements described in this chapter are: PROGRAM statement, INCLUDE statement, and END statement.

5.2 PROGRAM STATEMENT

This statement allows the user to give the main program a name other than "MAIN." The general form of a PROGRAM statement is

PROGRAM name

where

name	*is a user-formulated symbolic name that begins with an alphabetic character and contains a maximum of six characters. (Refer to section 3.3 for a description of symbolic names.)*

The following rule governs the use of the PROGRAM statement:

The PROGRAM statement must be the first statement in a program unit. (Refer to section 2.4 for a discussion of the ordering of DECsystem-20 FORTRAN statements.)

5.3 INCLUDE STATEMENT

This statement allows the user to include code segments or predefined declarations in a program unit without having them reside in the same physical file as the primary program unit. The general form of the INCLUDE statement is

INCLUDE dev:filename.type[proj,prog]/NOLIST

where

dev:	*is a device name. When no device is specified, DSK: is assumed.*
filename.type	*is the filename and type of the FORTRAN statements that the user wishes to include. The name of the file is required; the type is optional. If only the filename is specified, then .FOR (for FORTRAN) is the assumed type.*
[proj,prog]	*is the project-programmer number. The user's connected directory is assumed if none is specified. (Refer to Appendix B.)*

/NOLIST *is an optional switch that indicates that the included statements are not to be included in the compilation listing.*

The following rules govern the use of the INCLUDE statement:

a. The INCLUDEd file may contain any iegal FORTRAN statement except another INCLUDE statement, or a statement that terminates the current program unit, such as the END, PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statements.

b. The proper placement of the INCLUDE statement within a program unit depends upon the types of statements to be INCLUDEd. (Refer to section 2.4 for information on the ordering of DECsystem-20 FORTRAN statements.)

Note that an asterisk (*) is appended to the line numbers of the INCLUDEd statements on the compilation listing.

5.4 END STATEMENT

This statement is used to signal FORTRAN that the physical end of a source program or subprogram has been reached. END is a nonexecutable statement. The general form of an END statement is

```
END
```

The following rules govern the use of the END statement:

a. This statement must be the last physical statement of a source program or subprogram.

b. When used in a main program, the END statement implies a STOP statement operation; in a subprogram, END implies a RETURN statement operation.

c. An END statement may be labeled.

CHAPTER 6
SPECIFICATION STATEMENT

6.1 INTRODUCTION

Specification statements are used to specify the type characteristics, storage allocations, and data arrangement. There are seven types of specification statements:

a. DIMENSION

b. Statements which specify, explicitly, type.

c. *IMPLICIT*

d. COMMON

e. EQUIVALENCE

f. EXTERNAL

g. PARAMETER

Specification statements are nonexecutable and are expected to conform to the ordering guidelines described in section 2.4.

6.2 DIMENSION STATEMENT

DIMENSION statements provide FORTRAN with information needed to identify and allocate the space required for source program arrays. Any number of subscripted array names may be specified as array declarators in a DIMENSION statement. The general form of a DIMENSION statement is

DIMENSION S1, S2, . . ., Sn

where Si is an array declarator. Array declarators are names of the following form:

name (min:max, min:max, ... min:max)

where *name* is the symbolic name of the array and each min:max value represents the lower and upper bounds of an array dimension.

Each min:max value for an array dimension may be either an integer constant or, if the array is a dummy argument to a subprogram, an integer variable. The value given the minimum specification for a dimension must not exceed the value given the maximum specification. Minimum values of 1 with their following colon delimiter may be omitted from a dimension subscript.

Examples

```
DIMENSION EDGE (-1:1,4:8),NET(5,10,4),TABLE(567)
DIMENSION TABLE (IAB:J,K,M,10:20)
```

(where IAB, J, K, and M are of type integer).

Note that a slash may be used in place of a colon as the delimiter between the upper and lower bounds of an array dimension.

6.2.1 Adjustable Dimensions

When used within a subprogram, an array declarator may use type integer parameters as dimension subscript quantities. The following rules govern the use of adjustable dimensions in a subprogram:

a. For single entry subprograms, the array name and each subscript variable must be given by the calling program when the subprogram is called. The subscript variables may also be in COMMON.

b. For multiple entry subprograms in which the array name is a parameter, any subscript variables may be passed. If all subscript variables are not passed or in COMMON, the value of the subscript as passed for a previous entry will be used.

c. The type of the array dimension variables cannot be altered within the program.

d. If the value of an array dimension variable is altered within the program, the dimensionality of the array will not be affected.

e. The original size of the array cannot exceed the array dimensions assigned within a subprogram (i.e., the size of an array is not dynamically expandable).

Examples

```
      SUBROUTINE SBR (ARRAY,M1,M2,M3,M4)
      DIMENSION ARRAY (M1:M2,M3:M4)
      DO 27 L=M3,M4
      DO 27 K=M1,M2
      ARRAY (K,L)=VALUE
27    CONTINUE
      END

      SUBROUTINE SB1 (ARR1,M,N)
      DIMENSION ARR1(M,N)
      ARR1(M,N)=VALUE
      ENTRY SB2(ARR1,M)
      ENTRY SB3(ARR1,N)
      ENTRY SB4(ARR1)
```

In the foregoing example, the first call made to the subroutine must be made to SB1. Assuming that the call is made at SB1 with the values M=11 and N=13, any succeeding call to SB2 should give M a new value. If a succeeding call is made to SB4, the last values passed through entries SUB1, SUB2, or SUB3 will be used for M and N.

Note that for the calling program of the form:

```
CALL SB1(A,11,13)
M=15
CALL SB3(A,13)
```

the value of M used in the dimensionality of the array for the execution of SB3 will be 11 (i.e., the last value passed).

6.3 TYPE SPECIFICATION STATEMENTS

Type specification statements declare explicitly the data type of variable, array, or function symbolic names. An array name may be given in a type statement either alone (unsubscripted) to declare the type of all its elements or in a subscripted form to specify both its type and dimensions.

Type specification statements are written in the following form:

type list

where type may be any one of the following declarators:

a. INTEGER

b. REAL

c. DOUBLE PRECISION

d. COMPLEX

e. LOGICAL

NOTE

In order to be compatible with the type statements used by other manufacturers, the data type size modifier, *n, is accepted by DECsystem-20 FORTRAN. This size modifier may be appended to the declarators, causing some to elicit messages warning users of the form of the variable specified by DECsystem-20 FORTRAN:

Declarator	Form of Variable Specified
INTEGER*2	Full word integer with warning message
INTEGER*4	Full word integer
LOGICAL*1	Full word logical with warning message
LOGICAL*4	Full word logical
REAL*4	Full word real
REAL*8	Double precision real
COMPLEX*8	Complex
COMPLEX*16	Complex with warning message

NOTE (Cont)

In addition, the data type size modifier may be appended to individual variables, arrays, or function names. Its effect is to override, for the particular element, the size modifier (explicit or implicit) of the primary type. For example,

```
REAL*4 A, B*8, C*8(10), D
```

A and D are single precision (full word real), and B and C are double precision real.

The list consists of any number of variable, array, or function names which are to be declared the specified type. The names listed must be separated by commas, and can appear in only one type statement within a program unit.

Examples

```
INTEGER A, B, TABLE, FUNC
REAL R, M, ARRAY (5:10,10:20,5)
```

NOTE

Variables, arrays, and functions of a source program, which are not typed either implicitly or explicitly by a specification statement, are typed by FORTRAN according to the following conventions:

a. Variable names, array names, and function names which begin with the letters I, J, K, L, M, or N are type integer.

b. Variable names, array names, and function names which begin with any letter other than I, J, K, L, M, or N are type real.

If a name that is the same as a FORTRAN defined function name appears in a conflicting type statement, it is assumed that the name refers to a user-defined routine of the given type. Placing a generic FORTRAN defined function name in an explicit type statement causes it to lose its generic properties.

6.4 IMPLICIT STATEMENTS

IMPLICIT statements declare the data type of variables and functions according to the first letter of each variable name. IMPLICIT statements are written in the following form:

```
IMPLICIT type(A1,A2,. . .,An),type(B1,B2,. . .,Bn),. . .,type. . . . .
```

As shown in the foregoing form statement, an IMPLICIT statement is comprised of one or more type declarators separated by commas. Each type declarator has the form

```
type(A1,A2,. . .,An)
```

where type represents one of the declarators listed in section 6.3, and the parenthesized list represents a list of different letters.

Each letter in a type declarator list specifies that each source program variable (not declared in an explicit type specification statement) which starts with that letter is assigned the data type named in the declarator. For example, the IMPLICIT type declarator REAL (R,M,N,O) declares that all names which begin with the letters R, M, N, or O are type REAL names, unless declared otherwise in an explicit type statement.

NOTE

Type declarations given in an explicit type specification override those also given in an IMPLICIT statement. IMPLICIT declarations do not affect the DECsystem-20 FORTRAN supplied functions.

A range of letters within the alphabet may be specified by writing the first and last letters of the desired range separated by a dash (e.g., A–E for A,B,C,D,E). For example, the statement IMPLICIT INTEGER (I,L–P) declares that all variables which begin with the letters I,L,M,N,O, and P are INTEGER variables.

More than one IMPLICIT statement may be used, but they must appear before any other declaration statement in the program unit. Refer to section 2.4 for a discussion on ordering DECsystem-20 FORTRAN statements.

6.5 COMMON STATEMENT

The COMMON statement enables the user to establish storage which may be shared by two or more programs and/or subprograms and to name the variables and arrays which are to occupy the common storage. The use of common storage conserves storage and provides a means to implicitly transfer arguments between a calling program and a subprogram. COMMON statements are written in the following form:

```
COMMON/A1/V1,V2,. . .,Vn. . . /An/V1,V2,. . .,Vn
```

where the enclosed letters /A1/, /A2/, and /An/ represent optional name constructs (referred to as *common block names* when used).

The list (i.e., V1,V2. . .,Vn) appearing after each name construct lists the names of the variables and arrays that are to occupy the common area identified by the construct. The items specified for a common area are ordered within the storage area as they are listed in the COMMON statement.

COMMON storage area may be either labeled or blank (unlabeled). If the common area is to be labeled, a symbolic name must be given within slashes immediately before the list of items that are to occupy the names area. For example, the statement

```
COMMON/AREA1/A,B,C/AREA2/TAB(13,3,3)
```

establishes two labeled common areas (i.e., AREA1 and AREA2). Common block names bear no relation to internal variables or arrays which have the same name.

If a common area is to be declared but is to be unlabeled (i.e., blank) either nothing or two sequential slashes (//) is given immediately before the list of items that are to occupy blank common. For example, the statement

```
COMMON/AREA1/A,B,C//TAB(3,3,3)
```

establishes one labeled (AREA1) and one unlabeled (i.e., blank) common area.

A given labeled common name may appear more than once in the same COMMON statement and in more than one COMMON statement within the same program or subprogram.

Each labeled common area is treated as a separate, specific storage area. The contents of a common area (i.e., variables and array) may be assigned initial values by DATA statements in BLOCK DATA subprograms. Declarations of a given common area in different subprograms must contain the same number, size, and order of variable and array name as the referenced area.

Items to be placed in a blank common area may also be given in COMMON statements throughout the source program.

During compilation of a source program, DECsystem-20 FORTRAN strings together all items listed for each labeled common area and for blank common in the order in which they appear in the source program statements. For example, the series of source program statements

```
COMMON/ST1 /A,B,C/ST2/TAB(2,2)//C,D,E
.
.
COMMON/ST1/TST(3,4)//M,N
.
.
COMMON/ST2/X,Y,Z//O,P,Q
```

have the same effect as the single statement

```
COMMON/ST1/A,B,C,TST(3,4)/ST2/TAB(2,2),X,Y,Z//C,D,E,M,N,O,P,Q
```

All items specified for blank common are placed into one area. Items within blank common are ordered as they are given throughout the source program. Common block names must be unique with respect to all subroutine, function, and entry point names.

The largest definition of a given common area must be loaded first.

6.5.1 Dimensioning Arrays in COMMON Statements

Subscripted array names may be given in COMMON statements as array dimension declarators. However, variables cannot be used as subscript quantities in a declarator appearing in a COMMON statement; variable dimensioning is not permitted in COMMON.

Each array name given in a COMMON statement must be dimensioned either by the COMMON statement or by another dimensioning statement within the program or subprogram which contains the COMMON statement.

Example

```
COMMON /A/B(100), C(10,10)
COMMON X(5,15),Y(5)
```

6.6 EQUIVALENCE STATEMENT

The EQUIVALENCE statement enables the user to control the allocation of shared storage within a program or subprogram. This statement causes specific storage locations to be shared by two or more variables of either the same or different types. The EQUIVALENCE statement is written in the following form:

```
EQUIVALENCE(V1,V2,. . .Vn),(W1,W2,. . .Wn),(X1,X2,. . .)
```

where each parenthesized list contains the names of variables and array elements which are to share the same storage locations. For example, the statements

```
EQUIVALENCE (A,B,C)
EQUIVALENCE (LOC,SHARE(1))
```

specify that the variables named A, B, and C are to share the same storage location and the the variable LOC and array element SHARE(1) are to share the same location.

The relationship of equivalence is transitive; for example, the two following statements have the same effect:

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

Array elements, when used in EQUIVALENCE statements, must have either as many subscript quantities as dimensions of the array or only one subscript quantity. In either of the foregoing cases, the subscripts must be integer constants. Note that the single case treats the array as a one-dimensional array of the given type.

The items given in an EQUIVALENCE list may appear in both the EQUIVALENCE statement and in a COMMON statement providing the following rules are observed:

a. No two quantities declared in a COMMON statement can be set equivalent to one another.

b. Quantities placed in a common area by means of an EQUIVALENCE statement are permitted to extend the end of the common area forwards. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (A,Y)
```

cause the common block R to extend from Z to A(4) arranged as follows:

```
X
Y A(1)      (shared location)
Z A(2)      (shared location)
  A(3)
  A(4)
```

c. EQUIVALENCE statements that cause the start of a common block to be extended backwards are not allowed. For example, the invalid sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

would require A(1) and A(2) to extend the starting location of block R in a backwards direction as illustrated by the following diagram:

```
↑ A(1)
| A(2)
X A(3)
Y A(4)
Z
```

6.7 EXTERNAL STATEMENT

Any subprogram name to be used as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. The EXTERNAL statement declares names to be subprogram names to distinguish them from other variable or array names. The EXTERNAL statement is written in the following form:

```
EXTERNAL name1,name2,. . .,namen
```

where each name listed is declared to be a subprogram name. If desired, these subprogram names may be DECsystem-20 FORTRAN defined functions.

It is also possible to utilize DECsystem-20 FORTRAN defined function names for user subprograms by prefixing the names by an asterisk (*) or an ampersand (&) within an EXTERNAL statement. For example,

```
EXTERNAL *SIN, &COS
```

declares SIN and COS to be user subprograms. (If a prefixed name is not a DECsystem-20 FORTRAN defined function, then the prefix is ignored.)

Note that specifying a DECsystem-20 FORTRAN defined function in an EXTERNAL statement without a prefix (i.e., EXTERNAL SIN) has no effect upon the usage of the function name outside of actual argument lists. If the name has generic properties, they are retained outside of the actual argument list. (The name has no generic propeties within an argument list.)

The names declared in a program EXTERNAL statement are reserved throughout the compilation of the program and cannot be used in any other declarator statement, with the exception of a type statement.

6.8 PARAMETER STATEMENT

The PARAMETER statement allows users to define constants symbolically during compilation.

The general form of the PARAMETER Statement is as follows:

```
PARAMETER     P1=C1,P2=C2,. . . .
```

where

Pi — is a standard user-defined identifier (referred to in this section as a parameter name)

Ci — is any type of constant (including literals) except a label or complex constant. (Refer to Chapter 3 for a description of FORTRAN constants.)

During compilation the parameter names are replaced by their associated constants provided the following rules are observed:

a. Parameter names appear only within the statement field of an initial or continuation line type, i.e., not within a comment line or literal text.

b. Parameter names are placed only where FORTRAN constants are acceptable.

c. Parameter name references appear after the PARAMETER statement definition.

d. Parameter names are unique with respect to all other names in the program unit.

e. Parameter names are not redefined in subsequent PARAMETER statements.

f. Parameter names are not used as part of some larger syntactical construct (such as a Hollerith constant count, or a data type size modifier).

CHAPTER 7
DATA STATEMENT

7.1 INTRODUCTION

DATA statements are used to supply the initial values of variables, arrays, array elements, and labeled common.[1] DATA statements are written in the following form:

DATA List 1/Data 1/,List 2/Data 2/,. . .,List n/Data n/

where the List portion of each List/Data/ pair identifies a set of items to be initialized and the /Data/ portion contains the list of values to be assigned the items in the List. For example, the statement

DATA IA/5/,IB/10/,IC/15/

initializes variable IA as the value 5, variable IB as the value 10 and the variable IC as the value 15. The number of storage locations specified in the list of variables must be less than or equal to the number of storage locations specified in its associated list of values. If the list of variables is larger (specifies more storage locations) than its associated value list, a warning message is output. When the value list specifies more storage locations than the variable list the excess values are ignored.

The List portion of each List/Data/ set may contain the names of one or more variables, arrays, array elements, or labeled common variables. ***An entire array (unsubscripted array name) or a portion of an array may be specified in a DATA statement List as an implied DO loop construct (see Paragraph 10.3.4.1 for a description of implied DO loops). For example, the statement***

DATA (NARY (I), I=1,5)/1,2,3,4,5/

initializes the first five elements of array NARY as NARY(1)=1, NARY(2)=2, NARY(3)=3, NARY(4)=4, NARY(5)=5.

When an implied DO loop is used in a DATA statement, the loop index variable must be of type INTEGER and the loop Initial, Terminal, and Increment parameters must also be of type INTEGER. In a DATA statement, references to an array element must be integer expressions in which all terms are either integer constants or indices of embracing implied DO loops. Integer expressions of the foregoing types cannot include the exponentiation operator.

[1] Refer to Paragraph 6.5 for a description of labeled common.

The /Data/ portion of each List/Data/ set may contain one or more numeric, logical, ***literal,*** or ***octal*** constants and/or alphanumeric strings.

Octal constants must be identified as octal by preceding them with a double quote (") symbol (e.g., "777).

Literal data may be specified as either a Hollerith specification (e.g., 5HABCDE), or a string enclosed in single quotes (e.g., 'ABCDE'). Each ASCII datum is stored left-justified and is padded with blanks up to the right boundary of the variable being initialized.

When the same value is to be assigned to more than one item in List, a repeat specification may be used. The repeat specification is written as N*D where N is an integer that specifies how many times the value of item D is to be used. For example, a /Data/ specification of /3*20/ specifies that the value 20 is to be assigned to the first three items named in the preceding list. The statement

```
DATA M,N,L/3*20/
```

assigns the value 20 to the variables M, N, L.

In instances where the type of the data specified is not the same as that of the variable to which it is assigned, DECsystem-20 FORTRAN converts the datum to the type of the variable. The type conversion is performed using the rules given for type conversion in arithmetic assignments (refer to Chapter 8, Table 8-1). Octal, logical, and literal constants are not converted.

Sample Statement	Use
DATA PRINT,I,O/'TEST',30,"77/,TAB(J), J=1,30/30*5	The first 30 elements of array TAB are initialized as 5.0.
DATA ((A(I,J),I=1,5),J=1,6)/30*1.0/	No conversion required.
DATA ((A(I,J),I=5,10),J=6,15)/60*2.0/	No conversion required.

When a literal string is specified which is longer than one variable can hold, the string will be stored left-justified across as many variables as are needed to hold it. If necessary, the last variable used will be padded with blanks up to its right boundary.

Example

Assuming that X, Y, and Z are single precision, the statement

```
DATA X,Y,Z/'ABCDEFGHIJKL'/
```

will cause

X to be initialized to 'ABCDE'
Y to be initialized to 'FGHIJ'
Z to be initialized to 'KL␣␣␣'

When a literal string is to be stored in double precision and/or complex variables and the specified string is only one word long, the second word of the variable is padded with blanks.

Example

Assuming that the variable C is complex, the statement

```
DATA C/'ABCDE','FGHIJ'/
```

will cause the first word of C to be initialized to 'ABCDE' and its second word to be initialized to '␣␣␣␣␣'. The string 'FGHIJ' is ignored.

CHAPTER 8
ASSIGNMENT STATEMENTS

8.1 INTRODUCTION

Assignment statements are used to assign a specific value to one or more program variables. There are three kinds of assignment statements:

a. Arithmetic assignment statements

b. Logical assignment statements

c. Statement Label assignment (ASSIGN) statements.

8.2 ARITHMETIC ASSIGNMENT STATEMENT

Statements of this type are used to assign specific numeric values to variables and/or array elements. Arithmetic assignment statements are written in the form

v=e

where *v* is the name of the variable or array element which is to receive the specified value and *e* is a simple or compound arithmetic expression.

In assignment statements the equals symbol (=) does not imply equality as it would in algebraic expressions; it implies replacement. For example, the expression v=e is correctly interpreted as "the current contents of the location identified as *v* are to be replaced by the final value of expression e; the current contents of *v* are lost."

When the type of the specified variable or array element name differs from that of its assigned value, FORTRAN converts the value of the type of its assigned variable or array element. The type conversion operations performed by FORTRAN for each possible combination of variable and value types are described in Table 8-1.

Table 8-1
Rules for Conversion in Mixed Mode Assignments

Expression Type (e)	Variable Type (v)				
	Real	Integer	Complex	Double Precision	Logical
REAL	D	C	R,I	H,L	D
INTEGER	C	D	R,C,I	H,C,L	D
COMPLEX	R	C,R	D		R
DOUBLE PRECISION	H	C,H,L		D	H
LOGICAL	D	D	R,I	H,L	D,H
OCTAL	D	D	R,I	H,C,L	D
LITERAL	D,H***	C,H***	D**	D**	D***
DOUBLE OCTAL*	H	H	D****	D	H

Legend

D = Direct replacement
C = Conversion between integer and floating-point with truncation
R = Real part only
I = Set imaginary part to 0
H = High order only
L = Set low order part to 0

Notes

* *Octal numbers comprised of from 13 to 24 digits are termed double octal. Double octals require two storage locations. They are stored right-justified and are padded with zeroes to fill the locations.*

** *Use the first two words of the literal. If the literal is only one word long, the second word is padded with blanks.*

*** Use the first word of the literal.

**** *To convert double octal numbers to complex, the low order octal digits are assumed to be the imaginary part and the high order digits are assumed to be the real part of the complex value.*

8.3 LOGICAL ASSIGNMENT STATEMENTS

This type of assignment statement is used to assign values to variables and array elements of type logical. The logical assignment statement is written in the form

v=e

where *v* is one or more variables and/or array element names and *e* is a logical expression.

Examples

Assuming that the variables L, F, M, and G are of type logical, the following statements are valid:

Sample Statement	
L=.TRUE.	The contents of L are replaced by logical truth.
F=.NOT.G	The contents of L are replaced by the logical complement of the contents of G.
M=A > T	If A is greater than T, the contents of M are replaced by logical truth; if A is less than or equal to T, the contents of M are replaced by logical false.
L=((I.GT.H).AND.(J < =K))	The contents of L is replaced by either the true or false resultant of the expression.

8.4 ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT

The ASSIGN statement is used to assign a statement label constant (i.e., a 1- to 5-digit statement number) to a variable name. The ASSIGN statement is written in the following form

ASSIGN n TO I

where n represents the statement number and I is a variable name. For example, the statement

ASSIGN 2000 TO LABEL

specifies that the variable LABEL represents the statement number 2000.

With the exception of complex and double precision, any type of variable may be used in an ASSIGN statement.

Once a variable has been assigned a statement number, FORTRAN considers it a label variable. If a label variable is used in an arithmetic statement, the results are unpredictable.

The ASSIGN statement is used in conjunction with assigned GO TO control statements (Chapter 9); it sets up statement label variables which are then referenced in subsequent GO TO control statements. The following sequence illustrates the use of the ASSIGN statement:

```
555  TAX=(A+B+C)*.05
     .
     .
     .
     ASSIGN 555 TO LABEL
     .
     .
     .
     GO TO LABEL
```

DEC-system-20 FORTRAN extensions to the 1966
ANSI standard set are printed in *boldface italic type.*

CHAPTER 9
CONTROL STATEMENTS

9.1 INTRODUCTION

DECsystem-20 FORTRAN object programs are normally executed statement-by-statement in the order in which they were presented to the compiler. The following source program control statements, however, enable the user to alter the normal sequence of statement execution:

a. GO TO

b. IF

c. DO

d. CONTINUE

e. STOP

f. PAUSE

9.2 GO TO CONTROL STATEMENTS

There are three kinds of GO TO statements:

a. Unconditional

b. Computed

c. Assigned.

A GO TO control statement causes the statement which it identifies to be executed next, regardless of its position within the program. Each type of GO TO statement is described in the following paragraphs.

9.2.1 Unconditional GO TO Statements

GO TO statements of this type are written in the form

```
GO TO n
```

where n is the label (i.e., statement number) of an executable statement (e.g., GO TO 555). When executed, an unconditional GO TO statement causes control of the program to be transferred to the statement which it specifies.

An unconditional GO TO statement may be positioned anywhere in the source program except as the terminating statement of a DO loop.

9.2.2 Computed GO TO Statements

GO TO statements of this type are written in the form

```
GO TO (N1,N2,. . .,Nk)E
```

where the parenthesized list is a list of statement numbers and E is an arithmetic expression. Any number of statement numbers may be included in the list of this type of GO TO statement; however, each number given must be used as a label within the program or subprogram containing the GO TO statement.

NOTE
A comma may optionally follow the parenthesized list.

The value of the expression E must be reducible to an integer value that is greater than 0 and less than or equal to the number of statement numbers given in the statement's list. If E does not compute within the foregoing range, the next statement is executed.

When a computed GO TO statement is executed, the value of its expression (i.e., E) is computed first. The value of E specifies the position within the given list of statement numbers, of the number which identifies the statement to be executed next. For example, in the statement sequence

```
GO TO (20, 10, 5)K
CALL XRANGE(K)
```

the variable K acts as a switch causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3. The subprogram XRANGE is called if K is less than 1 or greater than 3.

9.2.3 Assigned GO TO Statements

GO TO statements of this type may be written in either of the following forms:

```
GO TO K
GO TO K, (L1,L2,. . .,Ln)
```

where K is a variable name and the parenthesized list of the second form contains a list of statement labels (i.e., statement numbers). The statement numbers given must be within the program or subprogram containing the GO TO statement.

Assigned GO TO statements of either of the foregoing forms must be logically preceded by an ASSIGN statement that assigns a statement label to the variable name represented by K. The value of the assigned label variable must be in the same program unit as the GO TO statement in which it is used. In statements written in the form

GO TO K, (L1,L2,. . .,Ln)

if K is not assigned one of the statement numbers given in the statement's list, then the next sequential statement is executed.

Examples

GO TO STAT1
GO TO STAT1, (177,207,777)

9.3 IF STATEMENTS

There are three kinds of IF statements: arithmetic, logical, and logical two-branch.

9.3.1 Arithmetic IF Statements

IF statements of this type are written in the form

IF (E) L1, L2, L3

where (E) is an expression enclosed within parenthesis and L1, L2, L3 are the labels (i.e., statement numbers) of three executable statements.

This type of IF statement causes control of the program to be transferred to one of the given statements, according to the computed value of the given expressions. If the value of the expression is:

a. less than 0, control is transferred to the statement identified by L1;

b. equal to 0, control is transferred to the statement identified by L2;

c. greater than 0, control is transferred to the statement identified by L3.

All three statement numbers must be given in arithmetic IF statements; the expression given may not compute to a complex value.

Examples

Sample Statement	
IF (ETA) 4, 7, 12	Transfer control to statement 4 if ETA is negative, to statement 7 if ETA is 0 and to statement 12 if ETA is greater than 0.
IF (KAPPA – L(10)) 20, 14, 14	Transfer control to statement 20 if KAPPA is less than the 10th element of array L and to statement 14 if KAPPA is greater than or equal to the 10th element of array L.

9.3.2 Logical IF Statements

IF statements of this type are written in the form

IF (E) S

where E is any expression enclosed in parentheses and S is a complete executable statement.

Logical IF statements cause control of the program to be transferred either to the next sequential executable statement or the statement given in the IF statement (i.e., S) according to the computed logical value of the given expression. If the value of the given logical expression is true (negative), control is given to the executable statement within the IF statement. If the value of the expression is false (positive or zero), control is transferred to the next sequential executable program statement.

The statement given in a logical IF statement may be any DECsystem-20 FORTRAN executable statement except a DO statement or another logical IF statement.

Examples

Sample Statement	
IF (T.OR.S) X = Y + 1	An arithmetic replacement operation is performed if the result of IF is true.
IF (Z.GT.X(K)) CALL SWITCH (S,Y)	A subprogram transfer is performed if the result of IF is true.
IF (K.EQ.INDEX) GO TO 15	An unconditional transfer is performed if the result of IF is true.

9.3.3 Logical Two-Branch IF Statements

IF statements of this type are written in the form

IF (E) N1, N2

where E is any expression enclosed in parentheses and N1 and N2 are statement labels defined within the program unit.

Logical two-branch IF statements cause control of the program to be transferred to either statement N1 or N2 depending on the computed value of the given expression. If the value of the given logical expression is true (negative), control is transferred to statement N1. If the value of the expression is false (positive or zero), control is transferred to statement N2.

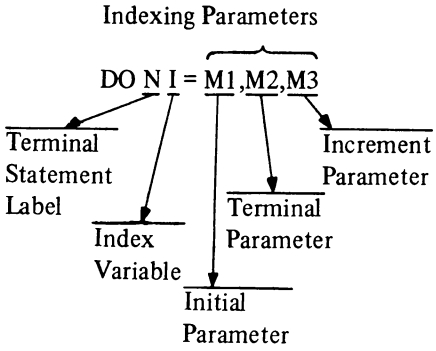
Note that the statement immediately following the logical two-branch IF must be numbered so that control can later be transferred to the portion of code that was skipped.

Examples

Sample Statement	
IF (LOG1) 10,20	*Transfer control to statement 10 if LOG1 is negative; otherwise transfer control to statement 20.*
IF (A.LT.B.AND.A.LT.C) 31, 32	*Transfer control to statement 31 if A is less than both B and C; transfer control to statement 32 if A is greater than or equal to either B or C.*

9.4 DO STATEMENT

DO statements simplify the coding of iterative procedures; they are written in the following form:



where

a. *Terminal Statement Label N* is the statement number of the last statement of the DO statement range. The range of a DO statement is defined as the series of statements which follows the DO statement up to and including its specified terminal statement.

b. *Index Variable I* is an unsubscripted variable, the value of which is defined at the start of the DO statement operations. The index variable is available for use throughout each execution of the range of the DO statement but its value should not be altered within this range. It is also made available for use in the program when

   1. control is transferred outside the range of the DO loop by a GO TO, IF, or RETURN statement located within the DO range,

   2. a CALL is executed from within the DO statement range which uses the index variable as an argument, and

   3. if an Input–Output statement with either or both the options END= or ERR= (Chapter 10) appear within the DO statement range.

c. *Initial Parameter M1* assigns the index variable, V, its initial value. This parameter may be any variable, array element, or expression.

d. *Terminal Parameter M2* provides the value which determines how many repetitions of the DO statement range are performed.

e. *Increment Parameter M3* specifies the value to be added to the initial parameter (M1) on completion of each cycle of the DO loop.

An indexing parameter may be any ***arithmetic expression*** which should result in either a positive or negative value. The values of the indexing parameters are calculated only once, at the start of each DO-loop operation. The number of times that a DO loop will be executed is specified by the formula:

(M2-M1)/M3+1

Since the count is computed at the start of a DO loop operation, changing the value of the loop index variable within the loop cannot affect the number of times that the loop is executed. At the start of a DO loop operation, the index value is set to the value of the initial parameter (M1) and a count variable (generated by the compiler) is set to the negative of the calculated count. At the end of each DO loop cycle the value of the increment parameter (M3) is added to the index variable and the count variable is incremented. If the number of specified iterations have not been performed, another cycle of the loop is initiated.

One execution of a DO loop range is always performed regardless of the initial values of the index variable and the indexing parameters.

Exit from a DO loop operation on completion of the number of iterations specified by the loop count is referred to as a normal exit. In a normal exit, control is passed to the first executable statement after the DO loop range terminal statement and the value of the DO statement index variable is considered undefined.

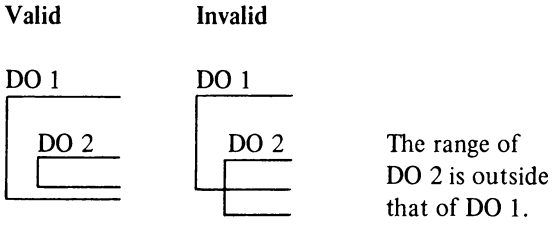
Exit from a DO loop may also be accomplished by a transfer of control by a statement within the DO loop range to a statement outside the range of the DO statement (Paragraph 9.4.3).

9.4.1 Nested DO Statements

One or more DO statements may be contained (i.e., nested) within the range of another DO statement. The following rules govern the nesting of DO statements.

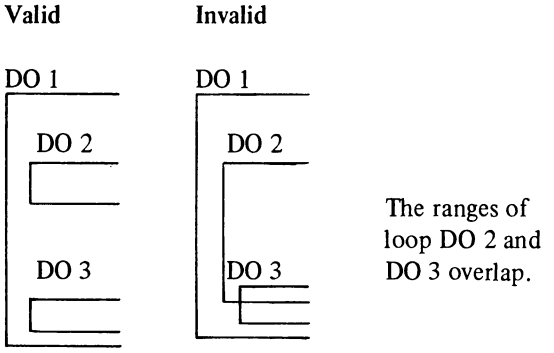
a. The range of each nested DO statement must be entirely within the range of the containing DO statement.

Example



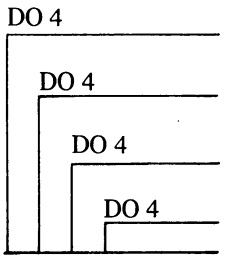
b. The ranges of nested DO statements cannot overlap.

Example



c. More than one DO loop within a nest of DO loops may end on the same statement. When this occurs, the terminal statement is considered to belong to the *innermost* DO statement that ends on that statement. The statement label 4 of the shared terminal statement cannot be used in any GO TO or arithmetic IF statement that occurs anywhere but within the range of the DO statement to which it belongs.

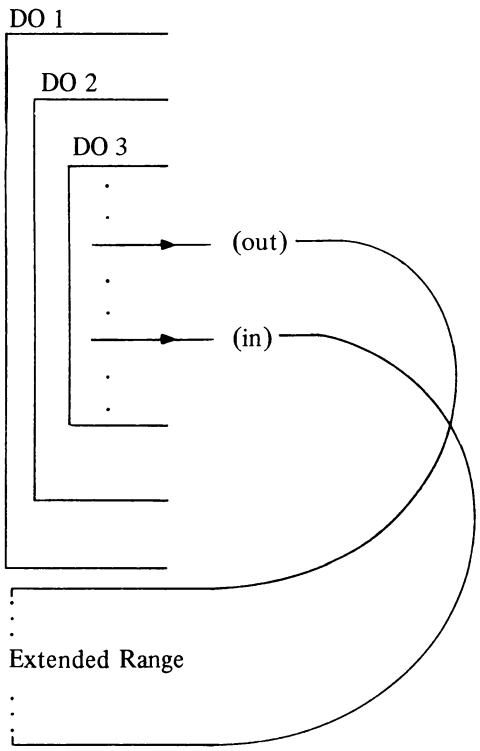
Example



All the DO statements share the same terminal statement, however, it belongs to DO 4.

9.4.2 Extend Range

The extended range of a DO statement is defined as the set of statements that are executed between the transfers out of the *innermost* DO statement of a set of nested DO's and the transfer back into the range of this innermost DO statement. The extended range of a nested DO statement is illustrated as follows:



The following rules govern the use of a DO statement extended range:

a. The transfer out statement for an extended range operation must be contained by the most deeply nested DO statement that contains the location to which the return transfer is to be made.

b. A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

c. The extended range of a DO statement must not contain another DO statement.

d. The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.

e. The use of and return from a subprogram from within an extended range is permitted.

9.4.3 Permitted Transfer Operations

The transfer of program control from within a DO statement range or the ranges of nested DO statements is governed by the following rules:

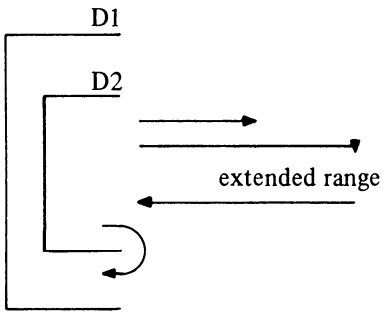
a. A transfer out of the range of any DO loop is permitted at any time. When such a transfer is executed the value of the controlling DO statement's index variable is defined as the current value.

b. A transfer into the range of a DO statement is permitted if it is made from the extended range of the DO statement.

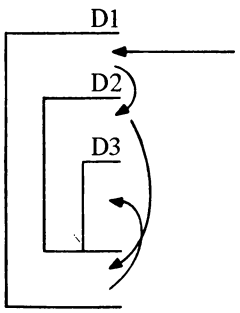
c. The use of and return from a subprogram from within the range of any DO loop, nested DO loop, or extended range is permitted.

The following examples illustrate the transfer operations permitted from within the ranges of nested DO statements.

Valid Transfers



Invalid Transfers



9.5 CONTINUE STATEMENT

CONTINUE statements may be placed anywhere in the source program without affecting the program sequence of execution. CONTINUE statements are commonly used as the last statement of a DO statement range in order to avoid ending with a GO TO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing any of the foregoing statements. This statement is written as

```
12    CONTINUE
```

Example

In the following sequence the labeled CONTINUE statement provides a legal termination for the range of the DO loop.

```
      .
      .
      DO 45 ITEM=1,1000
      STOCK=NVNTRY (ITEM)
      CALL UPDATE (STOCK,TALLY)
      IF (ITEM.EQ.LAST) GO TO 77
45    CONTINUE
      .
      .
      .
77    PRINT 20, HEADNG,PAGE NO
      .
      .
      .
```

9.6 STOP STATEMENT

When executed, the STOP statement causes the execution of the object program to be terminated and the user returned to command level. A descriptive message may, optionally, be included in the STOP statement to be output to the user's I/O terminal immediately before program execution is terminated. This statement may be written as

```
STOP
STOP 'N'
```

or

```
STOP n
```

where 'N' is a ***string of ASCII characters*** enclosed by single quotes and n is an ***octal string up to 12 digits.*** The string N or the value n is printed at the user's I/O terminal when the STOP statement is executed; it may be of any length, continuation lines may be used for large messages.

Examples

```
STOP 'Termination of the Program'
```

or

```
STOP 7777
```

9.7 PAUSE STATEMENT

When executed, a PAUSE statement causes a suspension of the execution of the object program and gives the user the option to:

a. Continue execution of the program

b. Exit

c. *Initiate a TRACE operation (Paragraph 9.7.1).*

The permitted forms of the PAUSE statement are:

a. PAUSE

b. PAUSE *'literal string'*

c. PAUSE n, where n is an *octal string up to 12 digits.*

The execution of a PAUSE statement of any of the foregoing forms causes the standard instruction:

TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

to be printed at the user's terminal. If the form of the PAUSE statement contains either a ***literal string*** or an integer constant, the string or constant is printed on a line preceding the standard message. For example, the statement

PAUSE *'TEST POINT A'*

causes the following to be printed at the user's terminal:

TEST POINT A
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

The statement

PAUSE 1

causes the following to be printed at the user's terminal:

PAUSE 000001
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

9.7.1 T (TRACE) Option

The entry of the character T in response to the message output by the execution of a PAUSE statement starts a TRACE routine. This routine causes the printing, at the user's terminal, of a complete history of all subroutine calls made during the execution of the program, up to the execution of the PAUSE statement. The history printed by the TRACE routine consists of:

a. The names of all subroutines called, arranged in the reverse order of their call;

b. The absolute location (written within parentheses) of the called subroutine;

c. The name of the calling subroutine plus an offset factor and the absolute location (written within parentheses) of the statement within the routine which initiated the call;

d. The number of arguments involved (written within angle brackets);

e. An alphabetic code (written within square brackets) that specifies the type of each argument involved. The alphabetic codes used and the meaning of each are:

Code Character	*Type Specified*
U	*Undefined type; the use of the argument will determine its type.*
L	*Logical*
I	*INTEGER*
F	*Single precision REAL*
O	*Octal*
S	*Statement Number*
D	*Double precision REAL*
C	*COMPLEX*
K	*A literal or constant*

Example

The following printout illustrates the execution of the PAUSE statement "PAUSE 'TEST POINT A'", the entry of a T character to initiate the TRACE routine, the resulting trace printout, and the entry of the character G to continue the execution of the program.

```
TEST POINT A
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE.
*T

NAME    (LOC)     <<---  CALLER  (LOC)   <#ARGS> [ARG TYPES]
TRACE.  (411653) <<---   MAIN.+612(1032)         <#1>     [U]
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE.
*G
```

In addition to its use with the PAUSE statement, the TRACE routine may be called directly, using the form

CALL TRACE

or as a function, using the form

X = TRACE (x)

Execution of the foregoing statements starts the TRACE routine which causes the printing of the history of all subprogram calls made during the execution of the program, up to the execution of the CALL statement, or up to the execution of the function, respectively. The history printed by the TRACE routine under these circumstances is exactly the same as described in the preceding paragraph.

DEC-system-20 FORTRAN extension to the 1966
ANSI standard set are printed in *boldface italic type.*

CHAPTER 10
I/O STATEMENTS

10.1 DATA TRANSFER OPERATIONS

FORTRAN I/O statements permit data to be transferred between processor storage (memory) and peripheral devices and/or between storage locations. Data in the form of logical records may be transferred using an a) sequential, b) ***random access,*** or c) ***append*** transfer mode. The areas in core from which data is to be taken during output (write) operations and into which data is stored during input (read) operations are specified by

a. a list in the I/O statement which initiated the transfer

b. ***a list defined by a NAMELIST statement, or***

c. between a specified FORMAT statement and the external medium.

The type and arrangement of transferred data may be specified by format specifications located in either a FORMAT statement or an array (formatted I/O) or by the contents of an I/O list ***(i.e., list-directed I/O).***

The transfer modes, I/O lists, type conversion and arrangement of data, and the statements required to initiate I/O transfer operations are described in the following paragraphs.

10.2 TRANSFER MODES

The characteristics and requirements of the a) sequential, b) ***random access,*** and c) ***append*** data modes are described in the following paragraphs.

10.2.1 Sequential Mode

Records are transferred during a sequential mode of operation in the same order as they appear in the external data file. Each I/O statement executed in a sequential mode transfers the record immediately following the last record transferred from the accessed source file.

10.2.2 Random Access Mode

This mode permits records to be accessed and transferred from a file in any desired order. Random access transfers, however, may be made only to (or from) a device that permits random-type data addressing operations (i.e., disk) and to files that have previously been set up for random access transfer operation. Files for random access must contain a specified number of identically sized records that may be accessed, individually, by a record number.

The OPEN statement or a subroutine call to DEFINE FILE may be used to set up random access files.

The OPEN statement is used to establish a random access mode to permit the execution of random access data transfer operations. The OPEN statement should logically precede the first I/O statement for the specified logical unit in the user source program.

10.2.3 Append Mode

This mode is a special version of the sequential transfer mode: it may be used only for sequential output (write) operations. The append mode permits the user to write a record immediately after the last logical record of the accessed file. During an append transfer, the records already in the accessed file remain unchanged, the only function performed is the appending of the transferred records to the end of the file.

An OPEN statement (Chapter 12) must be used to establish an append mode before append I/O operations can be executed.

10.3 I/O STATEMENTS, BASIC FORMATS AND COMPONENTS

The majority of the I/O statements described in this chapter are written in one of the following basic forms, or in some modification of these forms:

Basic Statement Forms	Use
Keyword (u,f)list	Formatted I/O Transfer
Keyword (u#R,f)list	***Random Access Formatted I/O Transfer***
Keyword (u,*)list	***List-Directed I/O Transfer***
Keyword (u,N)	***NAMELIST-Controlled I/O Transfer***
Keyword (u)list	Binary I/O Transfer
Keyword (u#R)list	Random Access Binary I/O Transfer

where

Keyword = the statement name (i.e., READ or WRITE)

u = logical unit number

f = FORMAT statement number or the name of an array that contains the desired format specifications

list = I/O list

#R = ***the delimiter # followed by the number of a record in an established random-access file***

******* = ***symbol specifying a list-directed I/O transfer.***

N = ***the name of an I/O list defined by a NAMELIST statement.***

Details of the foregoing statement components are given in the following paragraphs.

10.3.1 I/O Statement Keywords

The keywords (i.e., names) of the DECsystem-20 FORTRAN I/O statements described in this chapter are:

a. READ

b. REREAD

c. WRITE

d. ACCEPT

e. PRINT

f. TYPE

g. FIND

h. ENCODE

i. DECODE

j. DECODE

10.3.2 Logical Unit Numbers

The physical devices used for most FORTRAN I/O operations are identified by decimal numbers. During compilation, the compiler assigns default logical unit numbers for the *REREAD,* READ, *ACCEPT,* PRINT, and *TYPE* statements. Default unit numbers are negatively signed decimal numbers that are inaccessible to the user.

The logical device assignments may be made by the user at run time or the standard assignments contained by the FORTRAN Object Time System (FOROTS) may be used. The standard logical device assignments are listed in Table 10-1. It is recommended that the user specify the device explicitly in the OPEN statement.

10.3.3 FORMAT Statement References

A FORMAT statement contains a set of format specifications which define the structure of a record and the form of the data fields which comprise the record. Format specifications may also be stored in an array rather than in a FORMAT statement. (Refer to Chapter 13 for a complete description of the FORMAT statement.)

The execution of an I/O statement that includes either a FORMAT statement number or the name of an array which contains format specifications causes the structure and data of the transferred record to assume the form specified in the referenced statement or array. Records transferred under the control of a format specification are referred to as "formatted" records. Conversely, records transferred by I/O statements that do not reference a format specification are referred to as "unformatted" records. During unformatted transfers, data is transferred on a one-to-one correspondence between internal (processor) and external (device) locations, with no conversion or formatting operations.

Unformatted files are binary files divided into records by DECsystem-20 FORTRAN embedded control words; the control words are invisible to the user. Files of this type cannot be prepared by the user without utilizing FOROTS. Unformatted files are intended to be used only within the DECsystem-20 FORTRAN environment.

Table 10-1
DECsystem-20 FORTRAN Logical Device Assignments

Device/Function	*Default Filename*	*FORTRAN Logical Unit Number*	*Use*
*Standard Devices**			
0	*FORxx.DAT*	*00*	*ILLEGAL*
DSK		*01*	*DISK*
CDR		*02*	*Card Reader*
LPT		*03*	*Line Printer*
CTY		*04*	*Console Terminal*
TTY		*05*	*User's Terminal*
	06 through 15	*Not Valid*	
MTA0		*16*	*Magnetic Tape*
MTA1		*17*	
MTA2		*18*	
FORTR		*19*	*Assignable Device*
DSK		*20*	*DISK*
DSK		*21*	
DSK		*22*	
DSK		*23*	
DSK		*24*	
DEV1		*25*	*Assignable Devices*
DEV2		*26*	
DEV3		*27*	
DEV4		*28*	
DEV5		*29*	
DEV63	*FOR63.DAT*	*63*	*DISK*
Default Devices (inaccessible to the user)			
REREAD	*Current file*	*-6*	*REREAD statement*
CDR	*FORCDR.DAT*	*-5*	*READ statement*
TTY	*FORTTY.DAT*	*-4*	*ACCEPT statement*
LPT	*FORLPT.DAT*	*-3*	*PRINT statement*
		-2 Not Valid	
TTY	*FORTTY.DAT*	*-1*	*TYPE statement*

**The total number of standard devices permitted is on installation parameter.*

10.3.4 I/O List

An I/O list specifies the names of variables, arrays, and array elements to which input data is to be assigned or from which data is to be output. Implied DO constructs (Paragraph 10.3.4.1), which specify specific sets of array elements, may also be included in I/O lists. The number of items in a statement's list determines the amount of data to be transferred during each execution of the statement.

10.3.4.1 Implied DO Constructs – When an array name is given in an I/O list all elements of the array are transferred in the order described in Chapter 3 (Paragraph 3.5.3). If only a specific set of array elements is involved, they may be specified in the I/O list either individually or in the form of an implied DO construct.

Implied DO's are written within parentheses in a format similar to that of DO statements. They may contain one or more variable, array, and/or array element names, delimited by commas and followed by indexing parameters that are defined as for DO statements.

The general form of an implied DO is

(name(SL),I=M1,M2,M3)

where

name	=	an array name
SL	=	the subscript list of an array name or an array element identifier
I	=	the index control variable that represents a subscript appearing in a preceding subscript list
M1,M2,M3	=	the indexing parameters that specify, respectively, the initial, terminal, and increment values that control the range of I. If M3 is omitted (with its preceding comma), a value of 1 is assumed.

Examples

(A(S),S=1,5)	Specifies the first five elements of the one-dimension array A (i.e., A(1), A(2), A(3), A(4), A(5)).
(A(2,S),S=1,10,2)	Specifies the elements A(2,1), A(2,3), A(2,5), A(2,7), A(2,9) of array A.

As stated previously, implied DO constructs may also contain one or more variable names.

Example

I, J, B, and C must be integer variables.

((A(B,C),B=1,10),C=1,10),I,J	Specifies a 10 X 10 set of elements of array A, the location identified by I and the location identified by J.

Implied DO constructs may also be nested. Nested implied DO's may share one or more sets of indexing parameters.

Example

((A(J,K),J=1,5),D(K),K=1,10)	Specifies a 5 X 10 set of elements of array A and the first 10 elements of array D.

When an array or set of array elements are specified as either a storage or transmitting area for I/O purposes, the array elements involved are accessed in ascending order with the value of the first subscript quantity varying most rapidly and the value of the last given subscript increasing to its maximum value least rapidly. For example, the elements of an array dimensional as TAB(2,3) are accessed in the order:

TAB(1,1)
TAB(2,1)
TAB(1,2)
TAB(2,2)
TAB(1,3)
TAB(2,3)

10.3.5 The Specification of Records for Random Access

Records to be transferred in a random access mode must be identified in an I/O statement by an integer expression or variable preceded by a ' delimiter (e.g., '101).

NOTE

A number sign (#) may be used in place of the 'delimiter (e.g., both #101 and '101 are accepted by DECsystem-20 FORTRAN).

10.3.6 List-Directed I/O

The use of an asterisk in an I/O statement in place of a FORMAT statement number causes the specified transfer operation to be "list-directed." In a list-directed transfer, the data to be transferred and the type of each transferred datum are specified by the contents of an I/O list included in the I/O command used. The transfer of data in this mode is performed without regard for column, card, or line boundaries. The list-directed mode is specified by the substitution of an asterisk (*) for the FORMAT statement reference (i.e., f) of an I/O statement. The general form of a list-directed I/O statement is

keyword (u,*)list

Example

READ (5,*)I,IAB,M,L

List-directed transfers may be used to input data from any acceptable input device including an input keyboard terminal.

NOTE

Device positioning commands, such as BACKSPACE, SKIP RECORD, etc., should not be used in conjunction with list-directed I/O operations. If such a combination is used, the results will be unpredictable.

Data for list-directed transfers should consist of alternate constants and delimiters. The constants used should have the following characteristics:

a. ***Input constants must be of a type acceptable to DECsystem-20 FORTRAN. Octal constants, although acceptable, are not permitted in list-directed I/O operations.***

b. ***Literal constants must be enclosed within single quotes (e.g., 'ABLE').***

c. *Blanks serve as delimiters; therefore, they are not permitted in any but literal constants.*

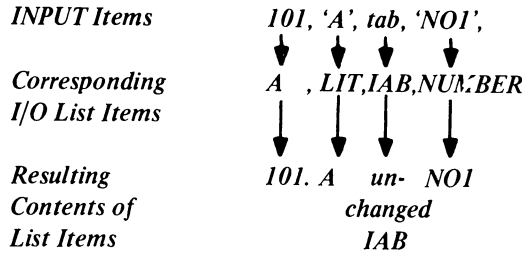
d. *Decimal points may be omitted from real constants which do not have a fractional part. FORTRAN assumes that the decimal point follows the right-most digit of a real constant.*

Delimiters in data for list-directed input must comply with the following:

a. *Delimiters may be either commas or blanks.*

b. *Delimiters may be either preceded by or followed by any number of blanks, carriage return/line feed characters, tabs, or line terminators; any such combination is considered by FORTRAN as being only a single delimiter.*

c. *A null, the complete absence of a datum, is represented by two consecutive commas which have no intervening constant(s). Any number of blanks, tabs, carriage return/line feed characters, or end-of-input conditions may be placed between the commas of a null. Each time a null item is specified in the input data, its corresponding list element is skipped (i.e., unchanged). The following illustrates the effect of a null input:*



d. *Slashes (/) cause the current input operation to be terminated even if all the items of the directing list are not filled. The contents of items of the directing I/O list which either are skipped (by null inputs) or have not received an input datum before the transfer is terminated remain unchanged. Once the I/O list of the controlling I/O statement is satisfied, the use of the / delimiter is optional.*

e. *Once the I/O list has been satisfied (transfers have been made to each item of the list) any items remaining in the input record are skipped.*

Constants or nulls in data for list-directed input may be assigned a repetition factor to cause an item to be repeated.

The repetition of a constant is written as

*r*K*

where r is an integer constant that specifies the number of times the constant represented by K is to be repeated.

The repetition of a null is written as an integer followed by an asterisk.

Examples

*10*5*	*represents 5,5,5,5,5,5,5,5,5,5*
3'ABLE'*	*represents 'ABLE','ABLE','ABLE'*
*3**	*represents null,null,null*

10.3.7 NAMELIST I/O Lists

One or more lists may be defined by a NAMELIST statement (Chapter 11). Each I/O list defined in a NAMELIST statement is identified by a unique (within the routine) 1 to 6 character name that may be referenced by one or more READ or WRITE statements. The first character of each I/O list name must be alphabetic. Referencing a NAMELIST-defined I/O list enables any of the foregoing statements to be written without an I/O list and permits the same list to be used by more than one statement.

I/O statements which reference a NAMELIST-defined I/O list cannot contain either a FORMAT statement reference or an I/O list. NAMELIST-controlled I/O operation cannot be used to transfer octal numbers or literal strings.

Records for NAMELIST-controlled input operations must be formatted in the following manner:

$NAME D1,D2,D3. . .Dn$

where

a. *$ symbols delimit the beginning and end of the record. The first $ must be in column 2 of the input record; column 1 must be blank.*

b. *NAME is the name of a NAMELIST-defined input list. The named list identifies the processor storage locations that are to receive the data items read from the accessed record.*

c. *D1 through Dn are values of the items of data contained by the record; these items cannot be octal numbers or literal strings.*

Only NAMELIST-controlled READ statements may be used to input records formatted in the foregoing manner.

NAMELIST-controlled WRITE statements will output records in the foregoing format.

NOTE

Device positioning commands such as BACKSPACE, SKIP RECORD, etc., should not be used in conjunction with NAMELIST-controlled I/O operations. If such a combination is used, the results are unpredictable.

10.4 OPTIONAL READ/WRITE ERROR EXIT AND END-OF-FILE ARGUMENTS

Either or both an error exit or an end-of-file argument may, optionally, be added to the parenthesized portion of most forms of the READ and WRITE I/O statements.

The error exit argument is written as ERR=c where c is a statement number. The use of this argument causes the current I/O operation to be terminated and program control transferred to the statement identified by the argument if a device error is detected. For example, the detection of an error during the execution of

READ(10,77,ERR=101)TABLE,I,M,J

terminates the input operation and transfers program control to statement 101.

The end-of-file argument is written as END=d where d is a statement number. The use of this argument causes the current I/O operation to be terminated and program control to be transferred to the statement identified by the argument when an end-of-file condition is detected. For example, the detection of an end-of-file condition during the execution of

READ(10,77,END=50)TABLE,I,M,J

transfers program control to statement 50.

If the END= argument is not present and an end of file (EOF) condition is detected, the file is closed, program execution is terminated, and the user is returned to command level.

10.5 READ STATEMENTS

READ statements transfer data from peripheral devices into specified processor storage locations. The permitted forms of this type of input statement permit READ statements to be used on both sequential and random access transfer modes for formatted, unformatted, list-directed, and NAMELIST-controlled data transfers.

10.5.1 Sequential Formatted READ Transfers

Descriptions of the READ statements that may be used for the sequential transfer of formatted data follow:

a. **Form:** READ (u,f)list

Use: Input data from logical unit *u*, formatted according to the specifications given in *f*, into the processor storage locations identified in input list.

Example: READ (10,555)TABLE(10,20),ABLE,BAKER,CHARL

b. **Form:** READ (u,f)

Use: Input the data from logical unit *u* directly into either a Hollerith (H) field descriptor or a literal field descriptor given within the format specifications of the referenced FORMAT statement. If the referenced FORMAT statement does not contain either of the foregoing types of format field descriptors, the input record is skipped. If a required field descriptor is present, its contents are replaced by the input data.

Example: READ(15,101)

c. **Form:** READ f

Use: Input the data from the READ default device (card reader) directly into either a Hollerith (H) field descriptor or a literal field descriptor given within the format specifications of the referenced FORMAT statement. If the referenced FORMAT statement does not contain either of the foregoing types of format field descriptors, the input record is skipped. If a required field descriptor is present, its contents are replaced by the input data.

Example: READ 66

d. **Form:** READ f, list

Use: Input the data from the READ default device (card reader) into the processor storage locations identified in the input list. The input data is formatted according to the specifications given in f.

Example: READ 15, ARRAY (20,30)

10.5.2 Sequential Unformatted Binary READ Transfers

Only the following form of the READ statement may be used for the sequential transfer of unformatted input FORTRAN binary data:

Form: READ (u)list

Use: Input one logical record of data from logical unit *u* into processor storage as the value of the location identified in list. Only binary files that have been output by a DECsystem-20 FORTRAN unformatted WRITE statement may be read by this type of READ statement.

NOTE
If the form READ (u) is used, it will cause one unformatted input record to be skipped.

Example: READ (10) BINFIL (10,20,30)

10.5.3 Sequential List-Directed READ Transfers

The following forms of the READ statements may be used to control a sequential, list-directed input transfer:

a. Form: *READ (u,*)list*

Use: *Input data from logical device u into processor storage or the value of the locations identified in list. Each input datum is converted, if necessary, to the type of its assigned list variable.*

Example: *READ (10,*) IARY (20,20), A,B,M*

b. Form: *READ *, list*

Use: *Input the data from the READ default device (card reader, CDR) into the processor storage locations identified in the input list. Each input datum is converted, if necessary, to the type of its assigned list variable.*

Example: *READ *, ABEL(10,20),I,J,K*

10.5.4 Sequential NAMELIST-Controlled READ Transfers

Only the following form of the READ statement may be used to initiate a sequential NAMELIST-controlled input transfer:

Form: *READ (u,n)*

Use: *Input data from logical unit u into processor storage as the value of the location identified by the NAMELIST input list specified by the name n. The input data is converted to the type of assigned variable if type conflicts occur. Only input files that contain records formatted and identified for NAMELIST operations (Paragraph 10.3.7) may be read by READ statements of this form.*

10.5.5 Random Access Formatted READ Transfers

Only the following form of the READ statement may be used to initiate a random access formatted input transfer:

Form: *READ (u#R,f)list*

Use: *Input data from record R of logical unit u. Format each input datum according to the format specifications of f and place into processor storage as values of the locations identified in list. Only disk files that have been set up by either an OPEN or DEFINE FILE statement may be accessed by a READ statement of this form. (If record R has not been written, a fatal error results.)*

10.5.6 Random Access Unformatted READ Transfers

Only the following form of the READ statement may be used to initiate a random-access unformatted input transfer:

Form: *READ (u#R)list*

Use: *Input data from record R of logical unit u. Place the input data into processor storage as the value of the locations identified in list. Only binary files that have been output by an unformatted random-access WRITE statement may be accessed by a READ statement of this form. (If record R has not been written, a fatal error results.)*

Example: *READ (1#20) BINFIL*

Read record number 20 into array BINFIL.

NOTE
If the form READ (u#R) is used, it will cause one logical input record to be skipped.

10.6 SUMMARY OF READ STATEMENTS

The various forms of the READ statements are summarized in Table 10-2.

Table 10-2
Summary of Read Statements

Type of Transfer	Transfer Mode	
	Sequential	Random Access
Formatted	READ (u,f)list READ (u,f) READ f,list READ f	READ (u#R,f)list
Unformatted	READ (u)list READ (u)	READ (u#R)list READ (u#R)
List-Directed	*READ (u,*)list* *READ * list*	
NAMELIST	*READ (u,N)*	

Note: The ERR=c and END=d arguments may be included in any of the above READ statements. When included, the foregoing arguments must be last, e.g., READ (10,20,END=101,ERR=500) ARRAY (50,100).

10.7 REREAD STATEMENT

The REREAD statement causes the last record read from the last active input device to again be accessed and processed.

The REREAD feature of DECsystem-20 FORTRAN cannot be used until an input (READ) transfer from a file has been accomplished. If REREAD is used prematurely, an error message will be output by DECsystem-20 FORTRAN at execution time.

Once a record has been accessed by a formatted READ statement the record transferred may be reread as many times as desired. In a formatted transfer, the same or new format specification may be used by each successive REREAD statement.

The REREAD statement may be used for sequential formatted data transfers only. The form of the REREAD statement is:

Form: *REREAD f,list*

Use: *Reread the last record read during the last initiated READ operation and input the data contained by the record into the processor storage locations specified in the input list. Format the data read according to the format specifications given in statement f.*

Example:

```
DIMENSION ARRAY(10,10),FORMA(10,10),FORMB(10,10),FORMC(10,10)
 90  READ(16,100)ARRAY
      .
      .
      .
100  FORMAT(_ _ _ _ _ _ _)
      .
      .
      .
110  REREAD 100,FORMA
115  REREAD 150,FORMB
120  REREAD 160,FORMC

150  FORMAT(_ _ _ _ _ _ _)
160  FORMAT(_ _ _ _ _ _ _)
```

In the above sequence, statement 90 inputs data formatted according to statement 100 into the array ARRAY. Statement 110 reads the record read by statement 90 and inputs the data formatted as in the initial READ operation into the array FORMA.

Statement 115 reads the record read by statements 90 and inputs the data formatted according to statement 150 into the array FORMB.

Statement 120 reads the record read by statement 90 and inputs the data formatted according to statement 160 into the array FORMC.

10.8 WRITE STATEMENTS

WRITE statements transfer data from specified processor storage locations to peripheral devices. The various forms of the WRITE statement enable it to be used in sequential, ***append and random access*** transfer modes for formatted, unformatted, ***list-directed*** and ***NAMELIST-controlled*** data transfers.

10.8.1 Sequential Formatted WRITE Transfers

The following forms of the WRITE statement may be used for the sequential transfer of formatted data:

a. **Form:** WRITE (u,f) list

Use: Output the values of the processor storage locations identified in list, into the file associated with logical unit *u*. Convert and arrange the output data according to the specifications given in statement or array *f*.

Example: WRITE(06,500)OUT(10,20),A,B

b. **Form:** WRITE f,list

Use: Output the values of the processor storage locations identified in list to the default device (i.e., line printer, LPT). Convert and arrange the output data according to the specifications given in f.

Example: WRITE 10, SEND(5,10),A,B,C

c. **Form:** WRITE f

Use: Output the contents of any Hollerith (H) or literal (") field descriptor(s) contained by f to the default device (i.e., line printer, LPT). If neither of the foregoing types of field specifications are found in f, no output transfer is performed.

Example: WRITE 10

10.8.2 Sequential Unformatted WRITE Transfer

The following form of the WRITE statements may be used for the sequential transfer of unformatted data:

Form: WRITE (u) list

Use: Output the values of the processor storage locations identified in list into the file associated with logical unit *u*. No conversion or arrangement of output data is performed.

Example: WRITE(12)ITAB(20,20),SUMS(10,5,2)

10.8.3 Sequential List-Directed WRITE Transfers

The following form of the WRITE statement may be used to initiate a sequential list-directed output transfer.

Form: *WRITE(u,*)list*

Use: *Output the values of the processor storage locations identified in list into the file associated with logical unit u. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.*

Example: *WRITE(12,*)C,X,Y,ITAB(10,10)*

10.8.4 Sequential NAMELIST-Controlled WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a sequential NAMELIST output transfer.

Form: *WRITE(u,N)*

Use: *Output the values of the processor storage locations identified by the contents of the NAMELIST-defined list specified by name N.*

Example: *WRITE(12,NMLST)*

10.8.5 Random Access Formatted WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a random access type formatted output transfer:

Form: *WRITE(u#R,f)list*

Use: *Output the values of the processor storage locations identified by the contents of list to record R of logical device u. Only disk files which have been set up by either an OPEN or a DEFINE FILE statement may be accessed by a WRITE transfer of this form. The data transferred will be formatted according to the specifications given in statement or array f Only those records which have been specifically written are available to be read.*

10.8.6 Random Access Unformatted WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a random access unformatted output transfer:

Form: *WRITE(u#R)list*

Use: *Output the values of the processor storage locations identified by the contents of list to record R of the logical device unit u. Only disk files which have been set up by either an OPEN or a call to the DEFINE FILE subroutine may be accessed by a WRITE transfer of this form. Only those records which have been specifically written are available to be read.*

10.9 SUMMARY OF WRITE STATEMENTS

The various forms of the WRITE statements are summarized in Table 10-3.

Table 10-3
Summary of WRITE Statements

Type of Transfer	Transfer Mode	
	Sequential	Random Access
Formatted	WRITE(u,f)list WRITE f,list WRITE f	WRITE(u#R,f)list
Unformatted	WRITE(u)list	WRITE(u#R)list
List-Directed	*WRITE(u,*)list*	
NAMELIST-controlled	*WRITE(u,N)*	

Note: *The ERR=c and END=d arguments may be included in any WRITE statement; however, they must be last.*

10.10 ACCEPT STATEMENT

The ACCEPT statement enables the user to input data via either a terminal keyboard or a Batch control file directly into specified processor storage locations. This statement is used only in the sequential transfer mode for the formatted transfer of inputs from the user's terminal keyboard during program execution. The permitted forms of the ACCEPT statement are described in the following paragraphs.

10.10.1 Formatted ACCEPT Transfers

The following forms of the ACCEPT statement are used for the sequential transfer of formatted data.

a. *Form:* *ACCEPT f,list*

Use: *Input data character-by-character into the processor storage locations identified by the contents of list. Format the input data according to the format specifications given in f.*

Example: *ACCEPT 101,LINE(73)*

b. *Form:* *ACCEPT *,list*

Use: ***Input data character-by-character into the processor storage locations identified by the contents of list. Convert the input characters, where necessary, to the type of its assigned list variable.***

Example: *ACCEPT *, IAB, ABE, KAB, MAR*

10.10.2 ACCEPT Transfers Into FORMAT Statement

The following form of the ACCEPT statement may be used to input data from the user's terminal keyboard directly into a specified FORMAT statement if the FORMAT statement has either or both a Hollerith (H) or literal ('s') field descriptor. If the referenced statement has neither of the foregoing field descriptors, the input record is skipped.

Form: *ACCEPT f*

Use: ***Replace the contents of the appropriate fields of statement f with the data entered at the user's terminal keyboard.***

Example: *ACCEPT 101*

10.11 PRINT STATEMENT

The PRINT statement causes data from specified processor storage locations to be output on the standard output device (i.e., line printer, LPT, Table 10-1). This statement may be used only for sequential formatted data transfer operation and may be written in either of the three following forms:

a. **Form:** PRINT f,list

Use: Output the values of the processor storage locations identified by the contents of list to the line printer. The values output are to be formatted and arranged according to the format specifications given in statement *f*.

Example: PRINT 55,TABLE(10,20),I,J,K

b. **Form:** PRINT *,list

Use: Output the values of the processor storage locations identified by the contents of list to the line printer. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.

Example: PRINT *,C,X,Y,ITAB(10,10)

c. **Form:** PRINT f

Use: Output the contents of the FORMAT statement Hollerith (H) or literal field descriptors to the line printer. If neither an H nor a literal field descriptor is present in the referenced FORMAT statement, no operation is performed.

Example: PRINT 55

The second form of the PRINT statement is particularly useful when employed with ACCEPT f statements to cause desired data (i.e., comments or headings) to be inserted into reports at program execution time.

Example

The sequence

```
55    FORMAT ('ϕENDϕOFϕROUTINE')
      .
      .
      .
      PRINT 55
```

results in the printing of the phrase END OF ROUTINE on the line printer.

10.12 TYPE STATEMENT

The TYPE statement causes data from specified processor storage locations to be output to the user's (control) terminal printing or display device (see Table 10-1 for device assignment for TYPE). This statement may be used only for sequential formatted data transfers and may be written in one of the following forms:

a. *Form:* ***TYPE f,list***

Use: ***Output the values of the processor storage locations identified by the contents of list to the user's terminal printing or display device. The values output are to be formatted according to the format specifications given in statement f.***

Example: ***TYPE 101,TABLE(10,20)I,J,K***

b. *Form:* ***TYPE f***

Use: ***Output the contents of the referenced FORMAT statement Hollerith (H) or literal field descriptors to the user's terminal printing or display device. If the referenced FORMAT statement does not contain either an H or a literal field descriptor, no operation is performed.***

Example: ***TYPE 101***

c. *Form:* ***TYPE *,list***

Use: ***Output the values of the processor storage locations identified by the contents of list to the printing or display device of the user's terminal. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.***

Example: ***TYPE *,IAB(1,5),A,B***

10.13 FIND STATEMENT

The FIND statement does not initiate a data transfer operation; it is used during random access read operations to locate the next record to be read while the current record is being input. The main program does not have access to the "found" record until the next READ statement is executed.

The form of the FIND statement is

FIND (u#R)

Example

In the sequence

```
READ (01#90)
FIND (01#101)
.
.
.
READ (01#101)
```

the FIND statement will locate record #101 on device 01 after record 90 has been retrieved. Record #101 is not actually processed until the second READ statement in the sequence is executed.

10.14 ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements are used to perform sequential formatted data transfer between two defined areas of processor storage (i.e., an I/O list and a user-defined buffer); no peripheral I/O device is involved in the operations performed by these statements.

The ENCODE statement transfers data from the variables of a specified I/O list into a specified user storage area. ENCODE operations are similar to those performed by a WRITE statement.

The DECODE statement transfers data from a specified user storage area into the processor storage locations identified by the variables of an I/O list. DECODE operations are similar to those performed by a READ statement.

The ENCODE and DECODE statements are written in the following forms:

```
ENCODE(c,f,s)list
DECODE(c,f,s)list
```

where

c specifies the number of characters to be in each internal storage area. This argument may be an integer, an integer expression, or either a real or double precision expression that is converted to an integer form.

> *NOTE*
> *Characters are stored in the buffer five characters per storage location without regard to the type of variable given as the starting location.*

f specifies either a FORMAT statement or an array that contains format specifications.

s specifies the address of the first storage location that is to be used in the transfer operations. When multiple records are specified by the format being used, the succeeding records follow each other in order of increasing storage addresses.

list specifies an I/O list of the standard form (Paragraph 10.3.4).

When multiple records are stored by ENCODE, each new record is started on a new boundary rather than there being a carriage return, line feed inserted between records.

10.14.1 ENCODE Statement

A description of the form and use of the ENCODE statement follows:

Form: *ENCODE(c,f,s)list*

Use: *The values of the processor storage locations identified by the contents of list are converted to ASCII character strings according to the format specifications contained by f. The converted characters are then written into the destination area starting at location s. If more characters are to be transferred than the specified area can contain, the excess characters are ignored; they are not written into any following records.*

If fewer characters are to be transferred than specified for the record size, the empty character locations are filled with blanks.

Example: *ENCODE(500,101,START)TABLE*

10.14.2 DECODE Statement

A description of the form and use of the DECODE statement follows:

Form: *DECODE(c,f,s)list*

Use: *The character strings stored in the internal reference and are read starting at location s, converted (decoded) according to the format specifications contained by f, and stored as the values of the locations identified in list.*

If the format specification requires more characters from a record than are specified by c, the extra characters are assumed to be blanks. If fewer characters are required from a record than are specified by c, the extra characters are ignored.

Example: *DECODE(50,50,START)GET(5,10)*

10.14.3 Example of ENCODE/DECODE Operations

The following program illustrates the use of both the ENCODE and DECODE statements:

Example

Assume the contents of the variables to be as follows:

A(1) contains the floating point binary number 300.45
A(2) contains the floating point binary number 3.0
J is an integer variable
B is a four-word array of indeterminate contents
C contains the ASCII string 12345

```
    DO 2 J = 1,2
    ENCODE(16,10,B) J, A(J)
10  FORMAT (1X,2HA(,I1,4H)=,F8.2)
    TYPE 11,B
```

```
11    FORMAT (4A5)
 2    CONTINUE
      DECODE (4, 12, C) B
12    FORMAT (3F1.0,1X,F1.0)
      TYPE 13,B
13    FORMAT (4F5.2)
      END
```

Array B can contain twenty ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

B(1)	*A(1)*	*Typed as*
B(2)	=	
B(3)	*300.4*	*A(1)=300.45*
B(4)	*5*	

The result after the second iteration is:

B(1)	*A(2)*	*Typed as*
B(2)	=	
B(3)	*3.0*	*A(2)=3.0*
B(4)		

The DECODE statement

a. extracts the digits 1, 2, and 3 from C

b. converts them to floating point binary value

c. stores them in B(1), B(2), and B(3)

d. skips the next character

e. extracts the digit 5 from C

f. converts it to a floating point binary value, and,

g. stores it in B(4).

10.15 SUMMARY OF I/O STATEMENTS

A summary of all permitted forms of the DECsystem-20 FORTRAN I/O statement is given in Table 10-4.

Table 10-4
Summary of DECsystem-20 FORTRAN I/O Statements

I/O Statements	Transfer Format Control			
	Formatted	Unformatted	*Namelist*	*List-Directed*
READ				
Sequential	READ(u,f)list READ f,list READ f	READ(u)list	*READ(u,n)*	*READ(u,*)list* *READ *,list*
Random	*READ(u#R,f)list*	*READ(u#R)list*		
WRITE				
Sequential or *Append*[1]	WRITE(u,f)list WRITE f,list WRITE f	WRITE(u)list	*WRITE(u,n)*	*WRITE(u,*)list*
Random[2]	*WRITE(u#R,f)list*	*WRITE(u#R)list*		
REREAD				
Sequential	*REREAD f,list*			
FIND				
Random-only	*FIND(u#R)*	*FIND(u#R)*		
ACCEPT				
Sequential only	*ACCEPT f,list* *or ACCEPT f*			*ACCEPT *,list*
PRINT				
Sequential only	PRINT f,list or PRINT f			*PRINT *,list*
TYPE				
Sequential only	*TYPE f,list* *or TYPE f*			*TYPE *,list*
ENCODE				
Sequential only	*ENCODE(c,f,s)list*			
DECODE				
Sequential only	*DECODE(c,f,s)list*			

Legend:

u	logical unit number	*	symbol used to specify list-directed I/O operator
f	statement number of FORMAT statement or name of array containing format information	#R	variable which specifies logical record position
list	I/O list	c	number of characters per internal record
n	name of specific NAMELIST I/O list	s	address of the first storage location to be used

[1] An OPEN statement must be used to set up an append mode.
[2] Either the OPEN statement or a call to the DEFINE FILE subroutine must be used to set up a random access mode.

DECsystem-20 FORTRAN extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 11
NAMELIST STATEMENTS

11.1 INTRODUCTION

The NAMELIST statement is used to define I/O lists similar to those described in Chapter 10 (Paragraph 10.3.4). Defined NAMELIST I/O lists are referenced in special forms of the READ and WRITE statements to provide a method of transferring and converting data without referencing format specifications or specifying an I/O list in the I/O statement.

11.2 NAMELIST STATEMENT

NAMELIST statements are written in the following form:

NAMELIST/N1/A1,A2,. . .,An/N2/B1,B2,. . .,Bn/Nn/. . .

where

/N/ through /Nn/	*represents names of individual lists; the names are always written enclosed by slashes (/N/)*
A1 through An and B1 through Bn	*are the items of the lists identified, respectively, by names N1 and N2. A list may contain one or more variable, array, or array element names. The items of a list are delimited by commas. Each list of a NAMELIST statement is identified (and referenced to) by the name immediately preceding the list.*

Example

NAMELIST/TABLE/A,B,C/SUMS/TOTAL

In the foregoing example, the name TABLE identifies the list A,B,C(2,4) and the name SUMS identifies the list comprised of the array TOTAL.

Once a list has been defined in a NAMELIST statement, its name may be referenced by one or more I/O statements.

The rules for structuring a NAMELIST statement are:

a. *A NAMELIST name may not be longer than six characters; it must start with an alphabetic character; it must be enclosed in slashes; it must precede the list of entries to which it refers; and it must be unique within the program.*

b. *A NAMELIST name may be defined only once and must be defined by a NAMELIST statement. Once defined, a name may appear only in READ or WRITE statements. The NAMELIST name must be defined in advance of the I/O statement in which it is used.*

c. *A variable used in a NAMELIST statement cannot be used as a dummy argument in a SUBROUTINE definition.*

d. *Any dimensioned variable contained in a NAMELIST statement must have been defined in a preceding array declaration statement.*

11.2.1 NAMELIST-Controlled Input Transfers

During input (read) transfer operations in which a NAMELIST-defined name is referenced, the record accessed is scanned until the symbol $ followed by the referenced name is found. Once the proper symbol-name combination is found, the data items following it are transferred on a one-to-one basis to the processor storage locations identified by the contents of the referenced list. The input data is always converted to the type of the list varible when there is a conflict of types. The input operation continues until another $ symbol is detected. If variables appear in the NAMELIST record that do not appear in the NAMELIST list, an error condition will occur. Data items of records to be input (read) using NAMELIST-defined lists must be separated by commas and may be of the following form:

V=K1,K2,. . .,Kn

where

a. *V may be a variable, array, or array element name.*

b. *K1 through Kn are constants of type integer, real, double precision, complex (written as (A,B) where A and B are real), or logical (written as T for true or F for false). A series of identical constants may be represented as a single constant preceded by a repetition factor (e.g., 5*5 represents 5,5,5,5,5).*

In transfers of this type, logical and complex constants must be equated to variables of their own type. Other type constants (real, double precision, and integer) may be equated to any other type of variable (except logical or complex), and will be converted to the variable type. For example, assume A is a 2-dimensional real array, B is a 1-dimensional integer array, C is an integer variable, and that the input data is as follows:

$FRED A(7,2)=4, B=3,6*2.8, C=3.32$

A READ statement referring to the NAMELIST defined name FRED will result in the following: the integer 4 will be converted to floating point and placed in A(7,2). The integer 3 will be placed in B(1) and the integer 2 (converted) will be placed in B(2),B(3),. . .,B(7). The floating point number 3.32 will be converted to the integer 3 and placed in C.

11.2.2 NAMELIST-Controlled Output Transfers

When a WRITE statement refers to a NAMELIST-defined name, all variables and arrays and their values belonging to the named list are written out, each according to its type. Arrays are written out by columns. Output data is written so that:

a. The fields for the data will be large enough to contain all the significant digits.

b. The output can be read by an input statement referencing a NAMELIST-defined list.

For example, if JOE is a 2 × 3 array, the statement

```
NAMELIST/NAM1/JOE,K1,ALPHA
WRITE (u,NAM1)
```

generates the following form of output:

```
Column
↓
$NAM1
JOE=   -6.75        .234E-04,        680,
       -17.8,          0.0          -.197E+07,
K1     =73.1,       ALPHA=3.$
```

DECsystem-20 FORTRAN extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 12
FILE CONTROL STATEMENTS

12.1 INTRODUCTION

File control statements are used to set up files and establish parameters for I/O operations and to terminate I/O operations.

The OPEN and CLOSE statements are described in this chapter.

12.2 OPEN AND CLOSE STATEMENTS

Both the OPEN and CLOSE statements use the same format and have the same options and arguments.

The OPEN statement enables the user to define, explicitly, all of the important aspects of each desired data transfer operation; they provide an extensive list of required and optional arguments which define in detail:

a. the name and location of the data file

b. the type of access required

c. the data format within the file

d. the protection code to be assigned an output data file

e. the disposition of the data file

f. data file record, block and file sizes

g. a data file version identifier

In addition, a DIALOG argument is provided which permits the user to establish a dialogue mode of operation when the OPEN statement containing it is executed. In a dialogue mode, interactive user terminal/program communication is established. This enables the user, during program execution, to define, redefine, or defer the values of the optional arguments contained by the current OPEN statement.

The general form of the OPEN statement is:

OPEN(Arg1,Arg2,. . .,Argn)

The CLOSE statement is used in the termination of an I/O operation to dissociate the I/O device being used from the active file and file-related information, and to restore the core occupied by I/O buffers and other transfer-related operations. All required device dependent termination functions are also performed on the execution of a CLOSE statement, including reloading the unit. Note that the CLOSE statement can change the name, and disposition of the file being closed.
Once a CLOSE statement has been executed, another OPEN statement is required to regain access to the closed file.

The general form of the CLOSE statement is:

CLOSE(Arg1.,Arg2.,. . .,Argn)

12.2.1 Options for OPEN and CLOSE Statements

The options and their arguments, which may be used in both the OPEN and CLOSE statements, are:

a. UNIT

This option is required; it defines the FORTRAN I/O unit number to be used. FORTRAN devices are identified by assigned decimal numbers within the range 1–63; however, UNIT may be assigned an integer variable or constant. The general form of this argument is:

UNIT = An integer variable or constant

DECsystem-20 FORTRAN standard logical unit assignments are described in Chapter 10 (Table 10-1). The range for the possible UNIT numbers is an installation defined parameter.

b. DEVICE

This option may specify either the physical or the logical name of the I/O device involved. (A logical name always takes precedence over a physical name.) The DEVICE arguments may specify I/O devices located at remote stations; as well as logical devices. The general form of the DEVICE argument is:

DEVICE = A literal constant or variable

If this option is omitted, the first logical name u (where u is the decimal unit number) is tried; if this is not successful, the standard (default) device is attempted.

c. ACCESS

A required option, ACCESS describes the type of input and/or output statements and the file access mode to be used in a specified data transfer operation. ACCESS may be assigned any one of six possible names, each of which specifies a specific type of I/O operation. The assignable names and the operations specified are:

1. SEQIN — *The specified data file is to be read in sequential access mode.*

2. SEQOUT — *The specified data file is to be written in a sequential access mode.*

3. SEQINOUT — *The specified data file may be first read then written (READ/WRITE sequence) record-by-record in a sequential access mode. When SEQINOUT is specified, a WRITE/READ sequence is illegal unless the file has been removed.*

4. RANDOM — *The specified data file may be either read or written into, one record at a time. In a random access mode of operation, the relative position of each record is independent of the previous READ or WRITE statement; all records accessed must have a fixed logical record length. This argument is required for random access operations. A disk device must be specified when the random argument is used.*

5. RANDIN — *This argument enables the user to establish a special, read-only random access mode with a named file. During a RANDIN mode, the user may read the named file simultaneously with other users who have also established a RANDIN mode and with the owner of the file. The use of RANDIN enables a data base to be shared by more than one user at the same time.*

6. APPEND — *The record specified by a corresponding WRITE statement is to be added to the logical end of a named file. The modified file must be closed then reopened in order to permit it to be read.*

The general form of the ACCESS argument is:

ACCESS =

'SEQIN'
'SEQOUT'
'SEQINOUT'
'RANDOM'
'RANDIN'
'APPEND'
variable (set to literal)

d. MODE

This option defines the character set of an external file or record. The use of this argument is optional; if it is not given, one of the following is assumed:

ASCII for a formatted I/O file transfer
Binary for an unformatted I/O file transfer

One of the following character set specifications must be used with the MODE argument:

Literal	*Action Indicated*
'ASCII'	*Specifies an ASCII character set.*
'BINARY'	*Specifies data formatted as a FORTRAN binary data file.*
'IMAGE'	*Specifies an image (I) mode data transfer for the associated READ or WRITE statements. IMAGE is an unformatted binary mode.*

The general form of the MODE argument is:

MODE =
'ASCII'
'BINARY'
'IMAGE'
variable (set to literal)

e. *DISPOSE*

This option specifies an action to be taken regarding a file at close time. When used, DISPOSE must be either an ASCII variable or one of the following literals:

Literal	*Action Indicated*
'SAVE'	*Leave the file on the device.*
'DELETE'	*If the device involved is disk, remove the file; otherwise, take no action.*
'PRINT'	*If the file is on disk, queue it for printing; otherwise, take no action.*
'LIST'	*If the file is on disk, queue it for printing and delete the file; otherwise take no action.*
'RENAME'	*Change filename. (This is redundant if a new filename is given.)*

If the DISPOSE argument is not given, the argument DISPOSE = SAVE is assumed. The general form of the DISPOSE argument is:

DISPOSE =
'SAVE'
'DELETE'
'PRINT'
'LIST'
'RENAME'
variable (set to literal)

f. FILE

This option specifies the name of the file involved in the data transfer operation. FILE must be either an ASCII literal, double precision, complex, or single precision variable. Single precision variables are assumed to contain a 1 to 5 character file specification; double precision variables, permit 10-character file specification. The format is a 1 to 6 character filename optionally followed by a period and a 0 to 3 character file type. Any excess characters in either the name or file type are ignored. If the period and file type are omitted, the file type .DAT is assumed; if just the file type is omitted, a "." is assumed.

If a file name is not specified or is zero, a default name is generated which has the form

FORxx.DAT

where xx is the FORTRAN logical unit number (decimal) or is the logical unit name for the default statements ACCEPT, PRINT, READ, or TYPE. The general form of a FILE argument is:

FILE = An ASCII literal or variable (set to literal)

g. PROTECTION

This option specifies a protection code to be assigned the data file being transferred. The protection code determines the level of access to the file that three possible classes of users (i.e., owner, member, or other) will have. PROTECTION may be a 3-digit octal literal or a variable; if the argument is assigned a zero value or is not given, the default protection code established for the DECsystem-20 installation is used. The general form of the PROTECTION argument is:

PROTECTION = 3-digit octal or integer variable

h. DIRECTORY

This option is used for disk files only. It specifies the location of the user file directory which contains the file specified in the OPEN statement. A directory identifier may consist of the user's project-programmer number for example, [10,7]. (Refer to Appendix B.)

The general form of a DIRECTORY argument is:

DIRECTORY= Literal or variable containing UFD name or directory path specification

i. BUFFER COUNT

This option enables the user to specify the number of I/O buffers to be assigned to a particular device. If this argument is not given or is assigned a value of zero, the Monitor default is assumed. The general form of this argument is:

BUFFER COUNT = An integer constant or variable

j. FILE SIZE

This option is used for disk operations only; it enables the user to estimate the number of words that an output file is going to contain. The use of FILE SIZE enables the user to ensure at the start of a program that enough space is available for its execution. If the size specified is found to be too small during program executions, the Monitor allocates additional space according to the normal Monitor algorithms. The value assigned to the FILE SIZE arguments may be an integer constant or variable. The general form of this argument is:

FILE SIZE = An integer constant or variable

k. VERSION

This option is used for disk operations only; it enables the user to assign a 12-digit octal version number to a file when it is output. The quantity assigned to the VERSION argument may be either an octal constant or variable. The general form of the argument is:

VERSION = An octal constant or integer variable

l. BLOCK SIZE

This option can be used for all storage media except disk. It enables the user to specify a physical storage block size for devices other than disk. The value assigned the BLOCK SIZE arguments may be an integer constant or variable. The size specified must be greater than or equal to 3 and less than or equal to 4095. The general form of this argument is:

BLOCK SIZE = An integer constant or variable

m. RECORD SIZE

This option enables the user to force all logical records to be a specified length. If a logical record exceeds the specified length, it is truncated; if a logical record is less than the specified size, nulls are added to pad the record to its full size. The RECORD SIZE argument is required whenever a random access mode is specified. The value assigned to this argument may be either an integer constant or variable, and may be expressed as the numbers of words or characters depending on the mode of the file being described. The general form of this argument is:

RECORD SIZE = An integer constant or variable

n. ASSOCIATE VARIABLE

This option is for disk random access operations only. It provides storage for the number of the record to be accessed next if the program being executed were to continue to access records one after another from the specified random access file. The general form of this argument is:

ASSOCIATE VARIABLE = Integer variable

o. PARITY

This option is for magnetic tape operations only; it permits the user to specify the type of parity to be observed (odd or even) during the transfer of data. The general form of this option is:

PARITY =	*'ODD'*
	'EVEN'
	variable (set to literal)

p. DENSITY

This option is for magnetic tape operations only; it permits the user to specify any of three possible bit-per-inch (bpi) tape density parameters for magnetic tape transfer operations. The general form of this option is:

	'200'
	'556'
DENSITY =	*'800'*
	'1600'
	variable (set to literal)

q. DIALOG

The use of this option in an OPEN statement enables the user to supersede or defer, at execution time, the values previously assigned to the arguments of the statement. There are two forms of this argument. The first is:

DIALOG

This form establishes a dialogue with the user's terminal when the OPEN statement is executed. FOROTS outputs the following messages at the user's terminal.

ENTER FILE SPECIFICATIONS FOR LOGICAL UNIT XX

(FOROTS then types the existing file specifications defined by the current OPEN statement.)

Once the message and defined file specification are output the user may enter any desired changes. Only the arguments that are to be changed need to be entered.

The second form of the argument is:

DIALOG =	*Literal or array*

The value assigned to DIALOG may be a literal or array containing a file specification with the desired information.

12.2.2 Summary of OPEN/CLOSE Statement Options

The options permitted and required in the OPEN and CLOSE statements and the type of value required by each are summarized in Table 12-1.

Table 12-1
OPEN/CLOSE Statement Arguments

Argument	*Values Required*
UNIT =	*Integer variable or constant*
MODE =	*Literal constant or variable*
DIRECTORY =	*Literal or variable*
FILE SIZE =	*Integer constant or variable*
BUFFER COUNT =	*Integer constant or variable*
ASSOCIATE VARIABLE =	*Integer variable*
ACCESS =	*'SEQIN', 'SEQOUT', 'SEQINOUT', 'RANDIN', 'RANDOM', 'APPEND', or variable*
FILE =	*Literal constant or variable*
DIALOG =	*Literal or array*
BLOCK SIZE =	*Integer constant or variable*
VERSION =	*Octal constant or variable*
DEVICE =	*Literal constant or variable*
PROTECTION =	*An octal constant or integer variable*
DISPOSE =	*Literal constant or variable*
RECORD SIZE =	*Integer constant or integer variable*
PARITY =	*Literal constant or variable*
DENSITY =	*Literal constant or variable*

CHAPTER 13
FORMAT STATEMENT

13.1 INTRODUCTION

FORMAT statements may appear almost anywhere in a FORTRAN source program. The only placement restric-
operations. The FORMAT statements contain field descriptors which, together with the list items of associated I/O statements, specify the forms of the data and data fields which comprise each record.

FORMAT statements may appear almost anywhere in a FORTRAN source program. The only placement restrictions are that they follow PROGRAM, FUNCTION, SUBPROGRAM, or BLOCK DATA statements, and that they precede the END statement. (Refer to Section 2.4.)

FORMAT statements must be labeled so that they can be referenced by I/O statements.

13.1.1 FORMAT Statement, General Form

The general form of a FORMAT statement follows:

k FORMAT(SA1,SA2,. . .,SAn/SB1,SB2,. . .,SBn/. . .)

where

k = the required statement label (which can only be referenced by I/O statements).

SA1 through *SAn* and *SB1* through *SBn* = individual field descriptor sets

In the foregoing statement form the individual field descriptors are delimited by commas (,) field descriptor sets and records are delimited by slashes (/). For example, a FORMAT statement of the form:

FORMAT(SA1,SA2/SB1,SB2/SC1,SC2)

contains format specifications for three records with each record comprised of two field descriptor sets.

Adjacent slashes (//) in a FORMAT statement specify that a record is to be skipped during input or is to consist of an empty record on output. For example, a FORMAT statement of the form:

```
FORMAT(SA1,SA2///SB1,SB2)
```

specifies four records are to be processed; however, the second and third records are to be skipped.

Repeated field descriptors or groups of field descriptors may be represented using a repeat form. The repetition of a single field descriptor is written by preceding the descriptor with an integer constant which specifies how many times the descriptor is to be repeated. For example, a FORMAT statement of the form

```
FORMAT(SA1,SA2,SA3,SA1,SA2,SA3,SA1,SA2,SA3)
```

may be written as

```
FORMAT(3(SA1,SA2,SA3))
```

The repeat forms of field descriptor may be nested to any depth. For example, a FORMAT statement of the form

```
FORMAT(SA1,SA2,SA2,SA3,SA1,SA2,SA2,SA3)
```

may also be written in the form

```
FORMAT(2(SA1,2SA2,SA3))
```

The manner in which the foregoing statement forms may be used and the effect each has on the data involved are discussed in the following paragraphs.

13.2 FORMAT DESCRIPTORS

FORMAT statement descriptors describe the record structure of the data, the format of the fields within the record, and the conversion, scaling, and editing of data within specific fields. The following descriptors can be used with DECsystem-20 FORTRAN:

Descriptors	Comments
rFw.d rEw.d rDw.d rGw.d	Floating point numeric field descriptors
rIw	Integer field descriptor
rLw	Logical field descriptor
rAw rRw	Alphanumeric data field descriptor
kHs 'text'	Alphanumeric data in a FORMAT statement field descriptor
rX *Tw*	Field formatting descriptors

Descriptors	Comments
nP	Numerical scale factor descriptor
/	Record delimiter
$	Carriage return suppression for terminal
rOw	Octal field descriptor

where

r = an optional unsigned integer that represents a repeat count. This option enables a field descriptor to be repeated *r* times.

w = an optional integer constant which represents the width (total number of characters contained) of the external form of the field being described. All characters including digits, decimal points, signs, and blanks that are to comprise the external form of the field must be included in the value of w.

.d = an optional unsigned integer that specifies the number of fractional digits which are to appear in the external representation of the field being described. Note that w must be specified if .d is included in the descriptor.

k = An unsigned integer that specifies the number of characters to be processed during the transfer of alphanumeric data.

s = represents a string of ASCII (alphanumeric) characters.

n = a signed integer constant (plus signs are optional).

The characters A, D, E, F, G, H, I, L, O, P, and R indicate the manner of conversion and editing to be performed between the internal (processor) and external representations of the data within a specific field; these characters are referred to as conversion codes. The DECsystem-20 FORTRAN conversion codes and a brief description of the function of each are given in Table 13-1.

Table 13-1
DECsystem-20 FORTRAN Conversion Codes

Code	Function
A	Transfer alphanumeric data
D	Transfer real data with a D exponent[1]
E	Transfer real data with an E exponent[1]
F	Transfer real data without an exponent
G	Transfer integer, real, complex, or logical data
H	Transfer literal data
I	Transfer integer data
L	Transfer logical data
O	***Transfer octal data***
P	Numerical scaling Factor
R	Transfer alphanumeric data

[1] An exponent of 0 is assumed if none is given.

The use of commas to delineate format descriptors within a format specification is optional as long as no ambiguity exists. For example,

```
FORMAT (3X,A2)
```

can be written as

```
FORMAT (3XA2)
```

Since interpretation of a format specification is left associative, the specification

```
FORMAT (I22,I5)
```

can be written as

```
FORMAT (I22I5)
```

However, a comma is required when the user wishes to specify

```
FORMAT (I2,2I5)
```

Detailed descriptions of the various types of format descriptors, the manner in which they are written and employed and their use in FORMAT statements are given in the following paragraphs.

13.2.1 Numeric Field Descriptors

The forms of the field descriptors used to specify the format and conversion of numeric data follow.

Description	Type of Data Used For
Dw.d	Double precision real data with a D exponent
Ew.d	Real data with an E exponent
Ew.d,Ew.d	For the real and imaginary parts of a complex datum
Fw.d	Real data without an exponent
Fw.d,Fw.d	For the real and imaginary parts of a complex datum
Iw	Integer data
Ow	Octal data
Gw.d	Real or double precision data
Gw	For integer (or logical) data
Gw.d,Gw.d	For the real and imaginary parts of a complex datum

NOTE

The G conversion code may be used for all but octal numeric data types.

Examples

Consider the following program segment:

```
INTEGER I1, I2
REAL R1, R2, R3
DOUBLE PRECISION D1, D2
I1 = 506
I2 = 8
R1 = 506.0
R2 = 18.1
R3 = 506001.0
D1 = 18.0
D2 = -504.0
.
.
.
```

The actions performed by several types of formatted WRITE statements on the data given in the foregoing program segment are described in Table 13-2.

Table 13-2
Action of Field Descriptors On Sample Data

Item	Descriptor Form	Sample Descriptor	WRITE Statement Using the Sample Descriptor	External Form of Sample Field Described	External Appearance of Sample Data
1	Dw.d	D8.2	WRITE (-,-) D1	Z.nnD±nn	0.18D+02
2	Ew.d	E8.2	WRITE (-,-) R1	Z.nnE±nn	0.51E+03
3	Fw.d	F5.2	WRITE (-,-) R2	aa.nnE+nn	18.10
4	Iw	I5	WRITE (-,-) I1	aaaan	␣␣506
5	Iw	I2	WRITE (-,-) I1	an	**
6	Ow	O5	WRITE (-,-) I2	nnnnn	00010
7	Gw.d	G8.2	WRITE (-,-) D2	Z.nnD±nn	-.50D+02
8	Gw.d	G8.2	WRITE (-,-) R3	Z.nnE±nn	0.51E+06
9	Gw.d	G8.2	WRITE (-,-) R2	aa.nn	18.10
10	Gw	G5	WRITE (-,-) I1	aaaan	␣␣506

where: a. *n* represents a numeric character

b. *Z* represents either a - or 0 (Note that if n-d > 6, a negative number cannot be output.)

c. *a* represents a digit, leading blank (␣) or a minus sign depending on the numeric output.

Notes:

1. In Item 1, the value D1 has only 2 significant digits and d=2, so no rounding will occur on input.

2. In Item 2, since R1 has 3 significant digits, it is rounded to fit into the specified field.

3. In Item 5, the width (*w*) part of a format descriptor specifies an exact field which permits no rounding of its contents. If the *w* specification is too small for the datum to be transferred, asterisks are output to indicate that the transfer was not made.

4. In Item 6, Integer 8 = Octal 10.

5. In Items 8 and 9, the relationship between G and fixed and floating real data is discussed in Paragraph 13.2.3.

6. In items 1, 2, 3, 7, and 8 the D and E exponent prefixes are optional in the external form of the floating point constants. For example, 1.1E+3 may be written as 1.1+3.

The internal and external forms of the data specified by the numeric format conversion code are summarized in Table 13-3.

Table 13-3
Numeric Field Codes

Internal Form	Conversion Code	External Form
Binary floating point double precision	D	Decimal floating point with D exponent
Binary floating point	E	Decimal floating point with E exponent
Binary floating point	F	Decimal fixed point
Binary integer	I	Decimal integer
Binary word	*O*	*Octal value*
One of the following: single precision, binary floating point, binary integer, binary logical, or binary complex	G	Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form

Complex quantities are transferred as two independent real quantities. The format specification for complex quantities consists of either two successive real field descriptors or one repeated real field descriptor. For example, the statement

```
FORMAT(2E15.4,2(F8.3,F8.5))
```

may transfer up to three complex quantities.

The equivalent of the foregoing statement is

```
FORMAT(E15.4,E15.4,F8.3,F8.5,F8.3,F8.5)
```

13.2.2 Interaction of Field Descriptors With I/O List Variables During Transfer

The execution of an I/O statement that specifies a formatted data transfer operation initiates format control. The actions performed by format control depend on information provided by the elements of the I/O statement's list of variables and the field descriptors which comprise the referenced FORMAT statement's format specifications.

In processing each FORMAT controlled I/O statement which has an I/O list, FORTRAN scans the contents of the list and the format specifications in step. Each time another variable or array element name is obtained from the list, the next field specification is obtained from the format specification. If the end of the format specification is reached and more items remain in the list, a new line or record is established and the scan process is restarted, either at the first item in the format specification or, if parenthesized sets of format specifications exist within the format specification, with the last set within the format specification.

When the I/O list is exhausted, control proceeds to the next statement in the program, but not before the FORMAT statement is scanned either to its end or to the next variable transfer format descriptor. (That is, the FORMAT statement is scanned past slashes, literal constants, and spacing descriptors, but not past data field descriptors.)

A record is terminated by one of the following:

a. a slash in the FORMAT specification

b. the delimiting right parentheses,), of the FORMAT statement

c. a lack of items in the I/O list

d. a lack of Hollerith field descriptors in the FORMAT statement

On input, an additional record is read only when a single slash, /, is encountered in the FORMAT statement. A record is skipped when two slashes, //, are encountered or a slash is followed by the end of the FORMAT statement. If the FORMAT statement finishes a record by a slash or the end of the FORMAT statement, then any data left in the input record is ignored. If the input record is exhausted before the data transfers are completed, the remainder of the transfer is completed as if the record were extended with blanks.

On output, an additional record is written only when a slash, /, is encountered in the FORMAT statement. If two consecutive slashes, //, or a single slash followed by the end of the FORMAT statement, is encountered, then an empty record is written.

13.2.3 G, General Numeric Conversion Code

The G conversion code may be used in field descriptors for the format control of real, double precision, integer, logical, or complex data.

With the exception of real and double precision data, the type of conversion performed by a G type field descriptor depends on the type of its corresponding I/O list variable. In the case of real and double precision data, the kind of conversion performed is a function of the external magnitude of the datum being transferred. Table 13-4 illustrates the conversions performed for various ranges of magnitude (external form) of real and double-precision data.

13.2.4 Numeric Fields with Scale Factors

Scale factors may be added to D,E,F, and G conversion codes in field descriptors. The scale factor has the form

nP

where *n* is a signed integer (+ is optional) and P identifies the operation. When used, a scale factor is added as a prefix to field descriptors.

Examples

-2PF10.5
1PE8.2

When added to an F type field descriptor (or G type if the external field is a fixed point decimal) a scale factor specifies a power of 10 so that

External Form of Number = (Internal Form)*$10^{(\text{scale factor})}$

For example, assuming the data involved to be the real number 26.451, the field descriptor

F8.3

produces the external field

ßß26.451

Table 13-4
Descriptor Conversion of Real and Double Precision Data According to Magnitude

Magnitude of Data in its External Form (M)	Equivalent Method of Conversion Performed
$0.1 \leqslant M < 1$	F(w-4).d,4X
$1 \leqslant M < 10$	F(w-4).(d-1),4X
.	.
.	.
.	.
.	.
$10^{d-2} \leqslant M < 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leqslant M < 10^{d}$	F(w-4).0,4X
ALL OTHERS	Ew.d

Note: In all numeric field conversions the field width (w) specified should be large enough to include the decimal point, sign, and exponent character in addition to the number of digits. If the specified width is too small to accommodate the converted number, the field will be filled with asterisks (*). If the number converted occupies fewer character positions than specified by *w*, it will be right-justified in the field and leading blanks will be used to fill the field.

The addition of the scale factor of -1P

```
-1PF8.3
```

produces the external field

```
␣␣␣2.645
```

When added to D, E, and G (external field not a decimal fixed point) type field descriptors, the scale factor multiplies the number by the specified power of ten and the exponent is changed accordingly.

In input operations, F type (and G type, if the external field is decimal fixed point) conversions are the only ones affected by scale factors.

When no scale factor is specified, it is understood to be zero. Once a scale factor is specified, however, it holds for all subsequent D, E, F, and G type field descriptors within the same format specification unless another scale factor is specified. A scale factor is reset to zero by specifying a scale factor of zero. Scale factors have no effect on I and O type field descriptors.

13.2.5 Logical Field Descriptors

Logical data may be transferred under format control in a manner similar to numeric data transfer by use of the field descriptor

Lw

where *L* is the control character and *w* is an integer specifying the field width. The data is transmitted as the value of a corresponding logical variable in the associated input/output list.

On input, the first non-blank character in the logical data field must be T or F, the value of the logical variable is stored in the list variable as true or false, respectively. If the entire input data field is blank or empty, a value of false is stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

13.2.6 Variable Numeric Field Widths

Several of the conversion codes are acceptable in FORMAT statements without field width specifications (i.e., the w.d portion of the specification is omitted[1]).

On input, the conversion codes D, E, F, G, I, L, and O are acceptable without field width specifications. The field begins with the first non-blank character encountered and ends with the first illegal character in the given field. (Blanks and tabs also terminate a field.) Note that for conversion code L (logical data) all consecutive alphabetics following a T (true) or an F (false) are considered part of the field and are ignored. In succeeding fields the input stream is scanned until a non-blank character is encountered. If the character is a comma (,) the next field is skipped and the following input field begins with the character following the comma. Any character other than a comma is assumed to be the first character in the next input field. Null fields are denoted by successive commas, optionally separated by blanks or tabs. A null field is equivalent to a fixed-field input of blanks. For example, the source code

```
   READ 1, X, Y, Z, L, I, J
1  FORMAT (3F, L, I, A3)
```

with data as follows

,1.0E+5,,TRUEXXX1␣␣␣␣ABC

results in

```
X = 0.0
Y = 1.0E+5
Z = 0.0
L = TRUE
I = 1
J = 'ABC'
```

Note that if a comma is included in the input data after the XXX1 and before the blanks, i.e., the data is

,1.0E+5 ,, TRUEXXX1,␣␣␣␣ABC

then J = '␣␣␣'

[1] If *d* is given, then w must also be specified.

On output, the format codes A, D, E, F, G, I, L, O, and R are acceptable without field width specifications. The following defaults are assumed:

Format Code	Assumed Default
A single precision	A5
A double precision	A10
D	D25.18
E	E15.7
F	F15.7
G single precision	G15.7
G double precision	G25.18
I	I15
L	L15
O	O15
R single precision	R5
R double precision	R10

13.2.7 Alphanumeric Field Descriptors

The formatted transfer of alphanumeric data may be accomplished in a manner similar to the formatted transfer of numeric data by use of the field descriptors Aw and Rw, where A and R are the control characters and w is the number of characters in the field.

The A and R descriptors both transfer alphanumeric data into or from a variable in an input/output list depending on the I/O operation. A list variable may be of any type. For example,

```
   READ (6,5) V
5  FORMAT (A4)
```

causes four alphanumeric characters to be read from the card reader and stored in the variable V.

The A descriptor deals with variables containing left-justified, blank-filled characters, and the R descriptor deals with variables containing right-justified, zero-filled characters. The following paragraphs summarize the result of alphanumeric data transfer (both internal and external representations) using the A and R descriptors. These paragraphs assume that w represents the field width and m represents the total number of characters possible in the variable. Double precision variables contain 10 characters (i.e., m=10); and all other variables contain 5 (i.e., m=5).

A Descriptor

a. INPUT, where $w \geq m$ – The rightmost m characters of the field are read in and stored left-justified and blank-filled in the associated variable.

b. INPUT, where $w < m$ – All w characters are read in and stored left-justified and blank-filled in the associated variable.

c. OUTPUT, where $w \geq m$ – m characters are output and right-justified in the field. The remainder of the field is blank-filled.

d. OUTPUT, where $w < m$ – The left-most w characters of the associated variable are output.

R Descriptor

a. INPUT, where w ⩾ m – The right-most m characters of the field are read in and stored right-justified, zero-filled in the associated variable.

b. INPUT, where w < m – All w characters are read in and stored right-justified, zero-filled in the associated variable.

c. OUTPUT, where w ⩾ m – m characters are output and right-justified in the field. The remainder of the field is blank filled.

d. OUTPUT, where w < m – The right-most w characters of the associated variable are output.

13.2.8 Transferring Alphanumeric Data Directly Into or From FORMAT Statements

Alphanumeric data may be transmitted directly into or from the FORMAT statement by two different methods: H-conversion, or the ***use of single quotes (i.e., a literal field descriptor).***

In H-conversion, the alphanumeric string is specified in the form nH, where *H* is the control character and *n* is the total number of characters (including blanks) in the string. For example, the following statement sequence may be used to print the words PROGRAM COMPLETE on the device LPT:

```
      PRINT 101
101   FORMAT (17HbPROGRAMbCOMPLETE)
```

Read and write operations of this type are initiated by I/O statements which reference a format statement and a logical device but do not contain an I/O list (see preceding example).

Write transfers from a FORMAT statement cause the contents of the statement field descriptor to be output to a specified logical device. The contents of the field descriptor, however, remain unchanged.

Read transfers with a FORMAT statement cause the contents of the field descriptors involved to be replaced by the characters input from the specified logical device.

Alphanumeric data is stored in a field descriptor left justified. If the data input into a field has fewer characters than the field, trailing blanks are added to fill the field. If the data input is larger than the field of the descriptor, the excess right most characters are lost.

Examples

```
      WRITE (1,101)
101   FORMAT (17HbPROGRAMbCOMPLETE)
```

cause the string PROGRAM COMPLETE to be output to the file on device 1.

Assuming the string START on device 1, the sequence

```
      READ (1,101)
101   FORMAT (17Hb PROGRAMbCOMPLETE)
```

would change the contents of statement 101 to

```
101 FORMAT (17HSTARTbbbbbbbbbbbbb)
```

The foregoing functions may also be accomplished by a ***literal field descriptor consisting of the desired character string enclosed within apostrophes (i.e., 'string'). For example, the descriptors***

```
101 FORMAT (17Hb PROGRAMbCOMPLETE)
```

and

```
101 FORMAT ('b PROGRAMbCOMPLETE')
```

may be used in the same manner.

The result of literal conversion is the same as H-conversion; on input, the characters between the apostrophes are replaced by input characters and, on output, the characters between the apostrophes (including blanks) are written as part of the output data.

An apostrophe character within a literal field should be represented by two successive apostrophe marks; otherwise, the statement containing the field will not compile. For example, the statement sequence

```
50 FORMAT ('DON''T')
   PRINT 50
```

will compile and will cause the word DON'T to be output on the line printer. The statement

```
50 FORMAT ('DON'T')
```

however, will cause a compile error.

13.2.9 Mixed Numeric and Alphanumeric Fields

An alphanumeric field descriptor may be placed among other fields of the format. For example, the statement:

```
FORMAT ( I4,7Hb FORCE=F10.5)
```

may be used to output the line:

```
bb22bFORCE=bb17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown in the foregoing statement.

When a comma delimiter is omitted from a format specification, format control associates as much information as possible with the leftmost of the two field descriptors.

13.2.10 Multiple Record Specifications

To handle a group of input/output records where different records have different field descriptors, a slash is used to indicate a new record. For example, the statement

```
FORMAT (308/I5,2F8.4)
```

is equivalent to

```
FORMAT (308)
```

for the first record, and

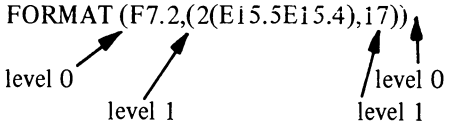
```
FORMAT (I5,2F8.4)
```

for the second record.

Separating commas may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records will be written on output or skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written on output or n-1 records skipped on input.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that the transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one or level zero if no level one group exists.

Thus, the statement



causes the format

```
2(E15.5,E15.4),I7
```

to be used on the first record.

As a further example, consider the statement

```
FORMAT (F7.2/(2(E15.5,E15.4),I7))
```

The first record has the format

```
F7.2
```

and successive records have the format

```
2(E15.5E15.4),I7
```

13.2.11 Record Formatting Field Descriptors

Two field descriptors, Tw and nX, may be used to position data within a record.

The field descriptor Tw may be used to specify the character position (external form) in which a record begins. In the Tw field descriptor the letter T is the control character and w is an unsigned integer constant which specifies the character position, in a DECsystem-20 FORTRAN record, where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position since the first character of the output buffer is a forms control character and is not printed. It is recommended that the first field specification of the output format be IX, except where a forms control character is used.

NOTE

Two successive T field specifications will result in the second field overwriting the first field.

Examples

The statement sequence

```
    PRINT 2
2   FORMAT (T50,'BLACK'T30,'WHITE')
```

causes the following line to be printed

```
      WHITE                BLACK
        ↑                    ↑
(print position 29)   (print position 49)
```

The statement sequence

```
1   FORMAT (T35,'MONTH')
    READ (2,1)
```

causes the first 34 characters of the input data associated with logical unit 2 to be skipped, and the next five characters to replace the characters M,O,N,T, and H in storage. If an input record containing

ABCbbbXYZ

is read with the format specification

10 FORMAT (T7,A3,T1,A3)

then the characters XYZ and ABC are read, in that order.

The field descriptor nX may be used to introduce blanks into output records or to skip characters of input records. The letter *X* specifies the operation and *n* is a positive integer that specifies the number of character positions to be either made blanks (output) or skipped (input).

Example

The statement

```
FORMAT (5HbSTEP15,10X2HY=F7.3)
```

may be used to print the line

```
STEPbbb28bbbbbbbbbbY=b-3.872
```

13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS

The first character of an ASCII record may be used to control the spacing operations of the line printer or terminal printer unit on which the record is being printed. The control character desired is specified by beginning the FORMAT field specification for the ASCII record to the output with 1HA . . . where *a* is the desired control character. The control characters permitted in DECsystem-20 FORTRAN and the effect each has on the printing device are described in Table 13-5.

Table 13-5
FORTRAN Print Control Characters

FORTRAN Character	Printer Character	Octal Value	Effect
space	LF	012	Skip to next line with form feed after 60 lines
0 zero	LF,LF	012	Skip a line
1 one	FF	014	Form feed – go to top of next page
+ plus			Suppress skipping – overprint the line
* asterisk	DC3	023	Skip to next line with no form feed
- minus	LF,LF,LF	012	Skip two lines
2 two	DLE	020	Space 1/2 of a page
3 three	VT	013	Space 1/3 of a page
/ slash	DC4	024	Space 1/6 of a page
. period	DC2	022	Triple space with a form feed after every 20 lines printed
, comma	DC1	021	Double space with a form feed after every 30 lines printed

Note: Printer control characters DLE, DC1, DC2, DC3, and DC4 affect only the line printer.

In order to print these control characters users must specify the switch /FILE:FORTRAN when giving the PRINT command.

CHAPTER 14
DEVICE CONTROL STATEMENTS

14.1 INTRODUCTION

The following device control statements may be used in FORTRAN source programs:

1. REWIND
2. ***UNLOAD***
3. BACKSPACE[1]
4. ENDFILE
5. ***SKIPRECORD***[1]
6. ***SKIPFILE,*** and
7. ***BACKFILE***

The general form of the foregoing device control statements is

keyword u
keyword (u)

where

keyword	is the statement name
u	is the FORTRAN logical device number (Chapter 10, Table 10-1)

The operations performed by the device control statement are normally used only for magnetic tape device (MTA). In DECsystem-20 FORTRAN, however, the device control operations are simulated for disk devices.

[1] The results of these commands are unpredictable when used on list-directed and NAMELIST-controlled data.

14.2 REWIND STATEMENT

Descriptions of the form and use of the REWIND statement follow:

Form: REWIND u

Use: Move the file contained by device *u* to its initial (load) point. If the medium is already at its load point, this statement has no effect. Subsequent READ or WRITE statements that reference device *u* will transfer data to or from the first record located on the medium mounted on device *u.*

Example: REWIND 16

14.3 UNLOAD STATEMENT

Descriptions of the form and use of the UNLOAD statement follow:

Form: ***UNLOAD u***

Use: ***Move the medium contained on device u past its load point until it has been completely rewound onto the source reel.***

Example: ***UNLOAD 16***

14.4 BACKSPACE STATEMENT

Descriptions of the form and use of the BACKSPACE statement follow:

Form: BACKSPACE u

Use: Move the medium contained on device *u* to the start of the record that precedes the current record. If the preceding record prior to execution of this statement was an endfile record, the endfile record becomes the next record after execution. If the current record is the first record of the file, this statement has no effect.

NOTE

This statement cannot be used for files set up for random access or NAMELIST-controlled I/O operations.

Example: BACKSPACE 16

14.5 END FILE STATEMENT

Descriptions of the form and use of the END FILE statement follow:

Form: END FILE u

Use: Write an endfile record in the file located on device *u.* The endfile record defines the end of the file which contains it. If an endfile record is reached during an I/O operation initiated by a statement that does not contain an END= option, the operation of the current program is terminated.

Example: END FILE 16

14.6 SKIP RECORD STATEMENT

Descriptions of the form and use of the SKIP RECORD statement follow:

Form: *SKIP RECORD u*

Use: *In accessing the file located on device u, skip the record immediately following the current (last accessed) record. The repeat option may be used to cause any desired number of records to be skipped.*

Example: *SKIP RECORD 16*

14.7 SKIP FILE STATEMENT

Descriptions of the form and use of the SKIP FILE statement follow:

Form: *SKIP FILE u*

Use: *In accessing the medium located on unit u, skip the file immediately following the current (last accessed) file. If the number of SKIP FILE operations specified exceeds the number of following files available, an error will occur.*

Example: *SKIP FILE 01*

14.8 BACKFILE STATEMENT

Descriptions of the form and use of the BACKFILE statement follow:

Form: *BACKFILE u*

Use: *Move the medium mounted on device u to the start of the file which precedes the current (last accessed) file.*

If the number of BACKFILE operations performed exceeds the number of preceding files, completion of the last operation will move the medium to the start of the first file on the medium.

Example: *BACKFILE 20*

14.9 SUMMARY OF DEVICE CONTROL STATEMENTS

The form and use of the DECsystem-20 FOTRAN device control statements are summarized in Table 14-1.

Table 14-1
Summary of DECsystem-20 FORTRAN Device Control Statements

Statement Form	Use
REWIND *u*	Rewind medium to its load point
UNLOAD u	*Rewind medium onto its source reel*
END FILE *u*	Write an endfile record in to the current file
SKIP RECORD u	*Skip the next record*
SKIP FILE u	*Skip the next file*
BACKFILE u	*Move medium backwards 1 file*
BACKSPACE *u*	Move medium back one record

CHAPTER 15
SUBPROGRAM STATEMENTS

15.1 INTRODUCTION

Procedures that are used repeatedly by a program may be written once and then referenced each time the procedure is required. Procedures that may be referenced are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that may be compiled separately). The kinds of FORTRAN procedures that may be referenced are:

a. statement functions

b. intrinsic functions (DECsystem-20 FORTRAN defined functions)

c. external functions, and

d. subroutines

The first three of the foregoing categories are referred to, collectively, as either functions or function procedures; procedures of the last category are referred to as either subroutines or subroutine procedures.

15.1.1 Dummy and Actual Arguments

Since subprograms may be referenced at more than one point throughout a program, many of the values used by the subprogram may be changed each time it is used. Dummy arguments in subprograms represent the actual values to be used which are passed to the subprogram when it is called.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments which they represent and whether the actual arguments are variables, array elements, arrays, subroutine names or the names of external functions. Each dummy argument must be used within a function or subroutine as if it were a variable, array, array element, subroutine, or external function identifier. Dummy arguments are given in an argument list associated with the identifier assigned to the subprogram; actual arguments are normally given in an argument list associated with a call made to the desired subprogram. (Examples of argument lists are given in the following paragraphs.)

The position, number, and type of each dummy argument in a subprogram list must agree with the position, number, and type of each actual argument given in the argument list of the subprogram reference.

Dummy arguments may be

a. variables

b. array names

c. subroutine identifiers

d. function identifiers, or

*e. statement label identifiers which are denoted by the symbol *, $, or &.*

When a subprogram is referenced, its dummy arguments are replaced by the corresponding actual arguments supplied in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the results of the subprogram computations will be unpredictable. Argument association may be carried through more than one level of subprogram reference if a valid association is maintained through each level. The dummy/actual argument associations established when a subprogram is referenced are terminated when the desired subprogram operations are completed.

The following rules govern the use and form of dummy arguments:

a. The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time the procedure is referenced.

b. Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.

c. A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding actual argument.

d. An array dummy argument should have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. Each element of a dummy array is associated directly with the corresponding elements of the actual array.

e. A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.

f. A dummy argument representing an external function must have an external function as its actual argument.

g. A dummy argument may be defined or redefined in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then elements of the array may be redefined.

Additional information regarding the use of dummy and actual arguments is given in the description of how subprograms are defined and referenced.

15.2 STATEMENT FUNCTIONS

Statement functions define an internal subprogram in a single statement. The general form of a statement function is:

name (arg1,arg2,. . .,argn)=E

where

name	is a user-formulated name comprised of from 1 to 6 characters. The name used must conform to the rules for symbolic names given in Paragraph 3.3. The type of a statement function is determined either by the first character of its name or by being declared in an explicit or implicit type statement.
(arg1. . .argn)	represents a list of dummy arguments.
E	is an arbitrary expression.

The expression *E* of a statement function may be any legitimate arithmetic expression which uses the given dummy arguments and indicates how they are combined to obtain the desired value. The dummy arguments may be used as variables or indirect function references; but they cannot be used as arrays. The dummy argument names bear no relation to their use outside the context of the statement function except for their data type. The expression may reference DECsystem-20 FORTRAN defined functions (Paragraph 15.3) or any other defined statement function, or call an external function. It may not reference any function that directly or indirectly references the given statement function or any subprogram in the chain of references. That is, recursive references are not allowed. Statement functions produce only one value, the result of the expression which it contains. A statement function cannot reference itself.

All statement functions within a program unit must be defined before the first executable statement of the program unit. When used, the statement function name must be followed by an actual argument list enclosed within parentheses and may appear in any arithmetic or logical expression.

Examples

```
SSQR(K)=(K*(K+1)*2*K+1)/6
ACOSH(X)=(EXP(X/A)+EXP(-X/A))/2.0
```

15.3 INTRINSIC FUNCTIONS (DECsystem-20 FORTRAN DEFINED FUNCTIONS)

Intrinsic functions are subprograms that are defined and supplied by DECsystem-20 FORTRAN. An intrinsic function is referenced by using its assigned name as an operand in an arithmetic or logical expression. The names of the DECsystem-20 FORTRAN intrinsic functions, the type of the arguments which each accepts, and the function it performs are described in Table 15-1. These names always refer to the intrinsic function unless they are preceded by an asterisk (*) or ampersand (&) in an EXTERNAL statement, declared in a conflicting explicit type statement, or are specified as a routine dummy parameter.

Table 15-1
Intrinsic Functions (DECsystem-20 FORTRAN Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Absolute value:					
Real	ABS*	arg	1	Real	Real
Integer	IABS*	arg	1	Integer	Integer
Double precision	DABS*	arg	1	Double	Double
Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real
Conversion:					
Integer to real	FLOAT*		1	Integer	Real
Real to integer	IFIX*	Sign of arg * largest integer $\leqslant \lvert arg \rvert$	1	Real	Integer
Double to real	SNGL		1	Double	Real
Real to double	DBLE*		1	Real	Double
Integer to double	DFLOAT		1	Integer	Double
Complex to real (obtain real part)	REAL*		1	Complex	Real
Complex to real (obtain imaginary part)	AIMAG		1	Complex	Real
Real to complex	CMPLX*	$c=Arg_1+i*Arg_2$	2	Real	Complex
Truncation:					
Real to real	AINT	Sign of arg * largest integer $\leqslant \lvert arg \rvert$	1	Real	Real
Real to integer	INT*		1	Real	Integer
Double to integer	IDINT		1	Double	Integer
Remaindering:					
Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real
Integer	MOD*		2	Integer	Integer
Double precision	DMOD		2	Double	Double
Maximum value:					
	AMAX0	$Max(Arg_1, Arg_2, \ldots)$	$\geqslant 2$	Integer	Real
	AMAX1*		$\geqslant 2$	Real	Real
	MAX0*		$\geqslant 2$	Integer	Integer
	MAX1		$\geqslant 2$	Real	Integer
	DMAX1		$\geqslant 2$	Double	Double
Minimum Value:					
	AMIN0	$Min(Arg_1, Arg_2, \ldots)$	$\geqslant 2$	Integer	Real
	AMIN1*		$\geqslant 2$	Real	Real
	MIN0*		$\geqslant 2$	Integer	Integer
	MIN1		$\geqslant 2$	Real	Integer
	DMIN1		$\geqslant 2$	Double	Double

*In line functions.

Table 15-1 (Cont)
Intrinsic Function (DECsystem-20 FORTRAN Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of	
				Argument	Function
Transfer of Sign:					
Real	SIGN*	$\left\{ Sgn(Arg_2)*\|Arg_1\| \right\}$	2	Real	Real
Integer	ISIGN		2	Integer	Integer
Double precision	DSIGN		2	Double	Double
Positive Difference:					
Real	DIM*	$\left\{ Arg_1 - Min(Arg_1, Arg_2) \right\}$	2	Real	Real
Integer	IDIM		2	Integer	Integer

*In line functions.

15.4 EXTERNAL FUNCTIONS

External functions are function subprograms that consist of a FUNCTION statement followed by a sequence of FORTRAN statements that define one or more desired operations; subprograms of this type may contain one or more RETURN statements and must be terminated by an END statement. Function subprograms are independent programs that may be referenced by other programs.

The FUNCTION statement that identifies an external function has the form

type FUNCTION name (arg1,arg2,. . .,argn)

where

type is an optional type specification as described in section 6.3. These include INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL (plus the optional size modifier, *n, for compatibility with other manufacturers.)

name is the name assigned to the function. The name may consist of from 1 to 6 characters, the first of which must be alphabetic. The optional size modifier (*n) may be included with the name if the type is specified. (Refer to section 6.3.)

(arg1,. . .,argn) is a list of dummy arguments.

If type is not given in the FUNCTION statement, the type of the function may be assigned, by default, according to the first character of its name, or may be specified by an IMPLICIT statement or by an explicit statement given within the subprogram itself.

Note that if a user wants to use the same name for a user-defined function as the name of a FORTRAN defined function (library basic external function), the desired name must be declared in an EXTERNAL statement and prefixed by an asterisk (*) or ampersand (&) in the referencing routine. (Refer to section 6.7 for a description of the EXTERNAL statement.)

The following rules govern the structuring of a FUNCTION subprogram:

a. The symbolic name assigned a FUNCTION subprogram must also be used as a variable name in the subprogram. During each execution of the subprogram this variable must be defined and, once defined, may be referenced as redefined. The value of the variable at the time of execution on any RETURN statement is the value of the subprogram.

NOTE

A RETURN statement returns control to the calling statement that initiated the execution of the subprogram. See Paragraph 15.4.1 for a description of this statement.

b. The symbolic name of a FUNCTION subprogram must not be used in any nonexecutable statement in the subprogram except in the initial FUNCTION statement or a type statement.

c. Dummy argument names may not appear in any EQUIVALENCE, COMMON, or DATA statement used within the subprogram.

d. The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

e. The function subprogram may contain any FORTRAN statement except BLOCK DATA, SUBROUTINE PROGRAM, another FUNCTION statement or any statement that directly or indirectly references the function being defined or any subprogram in the chain of subprograms leading to this function.

f. The function subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statement signifies a logical conclusion of the computation made by the subprogram and returns the computed function value and control to the calling program. A subprogram may have more than one RETURN statement.

The END statement specifies the physical end of the subprogram and implies a return.

15.4.1 Basic External Functions (DECsystem-20 FORTRAN Defined Functions)

DECsystem-20 FORTRAN contains a group of predefined external functions which are referred to as a basic functions. Table 15-2 describes each basic function, its name, and its use. These names always refer to the basic external functions unless declared in an EXTERNAL or conflicting explicit type statement.

15.4.2 Generic Function Names

The compiler generates a call to the proper DECsystem-20 FORTRAN defined function, depending on the type of the arguments, for the following generic function names:

ABS
AMAX1
AMIN1
ATAN
ATAN2
COS
INT
MOD
SIGN
SIN
SQRT
EXP
ALOG
ALOG10

In the following example

```
K=ABS (I)
```

the type of I determines which function is called. If I is an integer, the compiler generates a call to the function IABS. If I is real, the compiler generates a call to the function ABS. If I is double precision, the compiler generates a call to the function DABS.

The function name loses its generic properties if it appears in an explicit type statement, if it is specified as a dummy routine parameter, or if it is prefixed by "*" or "&" in an EXTERNAL statement. When a generic function name, which was specified unprefixed in an EXTERNAL statement, is used as a routine parameter, it is assumed to reference a DECsystem-20 FORTRAN defined function of the same name, or if none exist, a user-defined function. Note that IMPLICIT statements have no effect upon the data type of generic function names unless the name has been removed from its class using an EXTERNAL statement.

15.5 SUBROUTINE SUBPROGRAMS

A subroutine is an external computational procedure which is identified by a SUBROUTINE statement and may or may not return values to the calling program. The SUBROUTINE statement used to identify a subprogram of this type has the form:

SUBROUTINE *name*(arg1,arg2,. . .,argn)

where

name	is the symbolic name of the subroutine to be defined.
(arg1,. . .,argn)	is an optional list of dummy arguments.

Table 15-2
Basic External Functions (DECsystem-20 FORTRAN Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Exponential:					
Real	EXP	$\left\{ {}_{e}\text{Arg} \right\}$	1	Real	Real
Double	DEXP		1	Double	Double
Complex	CEXP		1	Complex	Complex
Logarithm:					
Real	ALOG	$\log_e(\text{Arg})$	1	Real	Real
	ALOG10	$\log_{10}(\text{Arg})$	1	Real	Real
Double	DLOG	$\log_e(\text{Arg})$	1	Double	Double
	DLOG10	$\log_{10}(\text{Arg})$	1	Double	Double
Complex	CLOG	$\log_e(\text{Arg})$	1	Complex	Complex
Square Root:					
Real	SQRT*	$(\text{Arg})^{1/2}$	1	Real	Real
Double	DSQRT	$(\text{Arg})^{1/2}$	1	Double	Double
Complex	CSQRT	$(\text{Arg})^{1/2}$	1	Complex	Complex
Sine:					
Real (radians)	SIN*	$\left\{ \sin(\text{Arg}) \right\}$	1	Real	Real
Real (degrees)	SIND		1	Real	Real
Double (radians)	DSIN		1	Double	Double
Complex	CSIN		1	Complex	Complex
Cosine:					
Real (radians)	COS*	$\left\{ \cos(\text{Arg}) \right\}$	1	Real	Real
Real (degrees)	COSD		1	Real	Real
Double (radians)	DCOS		1	Double	Double
Complex	CCOS		1	Complex	Complex
Hyperbolic:					
Sine	SINH	sinh(Arg)	1	Real	Real
Cosine	COSH	cosh(Arg)	1	Real	Real
Tangent	TANH	tanh(Arg)	1	Real	Real
Arc sine	ASIN	asin(Arg)	1	Real	Real
Arc cosine	ACOS	acos(Arg)	1	Real	Real
Arc tangent					
Real	ATAN*	atan(Arg)	1	Real	Real
Double	DATAN	datan(Arg)	1	Double	Double
Two REAL arguments	ATAN2*	$\text{atan}(\text{Arg}_1/\text{Arg}_2)$	2	Real	Real
Two DOUBLE arguments	DATAN2	$\text{atan}(\text{Arg}_1/\text{Arg}_2)$	2	Double	Double

*Generic Functions.

Table 15-2 (Cont)
Basic External Functions (DECsystem-20 FORTRAN Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of	
				Argument	Function
Complex Conjugate	CONJG	Arg=X+iY,CONJG=X-iY	1	Complex	Complex
Random Number	RAN	Result is a random number in the range of 0 to 1.0.	1 Dummy Argument	Integer, Real, Double, or Complex	Real

The following rules control the structuring of a subroutine subprogram:

a. The symbolic name of the subprogram must not appear in any statement within the defined subprogram except the SUBROUTINE statement itself.

b. The given dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement within the subprogram.

c. The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

d. The subroutine subprogram may contain any FORTRAN statement except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that either directly or indirectly references the subroutine being defined or any of the subprograms in the chain of subprogram references leading to this subroutine.

e. Dummy arguments that represent statement labels may be either an *, $, or &.

f. The subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statements indicate the logical end of a computational routine; the END statement signifies the physical end of the subroutine.

g. Subroutine subprograms may have as many entry points as desired (see description of ENTRY statement given in Paragraph 15.4.1).

15.5.1 Referencing Subroutines (CALL Statement)

Subroutine subprograms must be referenced using a CALL statement of the following form:

```
CALL name(arg1,arg2,. . .,argn)
```

where

name is the symbolic name of the desired subroutine subprogram.

(arg1,. . .,argn) is an optional list of actual arguments. If the list is included, the given actual arguments must agree in order, number, and type with the corresponding dummy arguments given in the defining SUBROUTINE statement.

The use of literal constants is an exception to the rule requiring agreement of type between dummy and actual arguments. An actual argument in a CALL statement may be:

a. a constant

b. a variable name

c. an array element identifier

d. an array name

e. an expression

f. the name of an external subroutine, or

g. a statement label.

Example

The subroutine

```
SUBROUTINE MATRIX(I,J,K,M,*)
.
.
.
END
```

may be referenced by

```
CALL MATRIX(10,20,30,40,$101)
```

15.5.2 DECsystem-20 FORTRAN Supplied Subroutines

DECsystem-20 FORTRAN provides the user with an extensive group of predefined subroutines. The descriptions and names of these predefined subroutines are given in Table 15-3.

15.6 RETURN STATEMENT AND MULTIPLE RETURNS

The RETURN statement causes control to be returned from a subprogram to the calling program unit. This statement has the form

```
RETURN    (standard return)
```

or

```
RETURN e    (multiple returns)
```

where e represents an integer constant, variable, or expression. The execution of this statement in the first of the foregoing forms (i.e., standard return) causes control to be returned to the statement of the calling program which follows the statement that called the subprogram.

The multiple returns form of this statement (i.e., RETURN e) enables the user to select any labeled statement of the calling program as a return point. When the multiple returns form of this statement is executed, the assigned or calculated value of e specifies that the return is to be made to the eth statement label in the argument list of the calling statement. The value of e should be a positive integer which is equal to or less than the number of statement labels given in the argument list of the calling statement. If e is less than 1 or is larger than the number of available statement labels, a standard return operation is performed.

NOTE

A dummy argument for a statement label must be either a *, $, or & symbol.

Any number of RETURN (standard return) statements may be used in any subprogram. The use of the multiple returns form of the RETURN statement, however, is restricted to SUBROUTINE subprograms. The execution of a RETURN statement in a main program will terminate the program.

Example

Assume the following statement sequence in a main program:

```
        .
        .
        .
        CALL EXAMP(1,$10,K,$15,M,$20)
        GO TO 101
        .
        .
        .
10      . . . . . .
        .
        .
        .
15      . . . . . .
        .
        .
        .
20      . . . . . .
        .
        .
        .
```

Assume the following statement sequence in the called SUBROUTINE subprogram:

```
SUBROUTINE EXAMP (L,*,M,*,N,*)
.
.
RETURN
.
.
RETURN
.
.
RETURN(C/D)
.
.
END
```

Each occurrence of RETURN returns control to the statement GO TO 101 in the calling program.

If, on the execution of the RETURN(C/D) statement, the value of (C/D) is:

Less than or equal to:	*The following is performed:*
0	*a standard return to the GO TO 101 statement is made*
1	*the return is made to statement 10*
2	*the return is made to statement 15*
3	*the return is made to statement 20*

Greater than or equal to:	*The following is performed:*
4	*a standard return to the GO TO 101 statement is made.*

15.6.1 Referencing External FUNCTION Subprograms

An external function subprogram is referenced by using its assigned name as an operand in an arithmetic or logical expression in the calling program unit. The name must be followed by an actual argument list. The actual arguments in an external function reference may be:

a. a variable name

b. an array element identifier

c. an array name

d. an expression

e. a statement number

f. the name of another external procedure (FUNCTION or SUBROUTINE).

NOTE

Any subprogram name to be used as an argument to another subprogram must first appear in an EXTERNAL statement (Chapter 6) in the calling program unit.

Example

The subprogram defined as:

```
INTEGER FUNCTION ICALC(X,Y,Z)
.
.
.
RETURN
END
```

may be referenced in the following manner:

```
.
.
TOTAL = ICALC(IAA,IAB,IAC)+500
```

15.7 MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT)

DECsystem-20 FORTRAN provides an ENTRY statement which enables the user to specify additional entry points into an external subprogram. This statement used in conjunction with a RETURN statement enables the user to employ only one computational routine of a subprogram which contains several such routines. The form of the ENTRY statement is:

ENTRY name(arg1,arg2,. . .,argn)

where

name — *is the symbolic name to be assigned the desired entry point*

(arg1,. . .,argn) — *is an optional list of dummy arguments. This list may contain*

a. *variable names*

b. *array declarators*

c. *the name of an external procedure (SUBROUTINE or FUNCTION), or*

d. *an address constant denoted by either a *, $, or & symbol*

The rules for the use of an ENTRY statment follow.

a. *The ENTRY statement allows entry into a subprogram at a place other than that defined by the subroutine or function statement. Any number of ENTRY statements may be included in an external subprogram.*

b. *Execution is begun at the first executable statement following the ENTRY statement.*

c. *Appearance of an ENTRY statement in a subprogram does not preclude the rule that statement functions in subprograms must precede the first executable statement.*

d. *Entry statements are nonexecutable and do not affect the execution flow of a subprogram.*

e. *An ENTRY statement may not appear in a main program, nor may a subprogram reference itself through its entry points.*

f. *An ENTRY statement may not appear in the range of a DO or an extended DO statement construction.*

g. *The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in SUBROUTINE or FUNCTION statements or any other ENTRY statement in the subprogram. However, the arguments for each call or function reference must agree with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that is referenced.*

h. *Entry into a subprogram initializes the dummy arguments of the referenced ENTRY statement, all appearances of these arguments in the entire subprogram are initialized.*

i. *A dummy argument may not be referenced unless it appears in the dummy list of an ENTRY, SUBROUTINE, or FUNCTION statement by which the subprogram is entered.*

j. The source subprogram must be ordered such that references to dummy arguments in executable statements must follow the appearance of the dummy argument in the dummy list of a SUBROUTINE, FUNCTION, or ENTRY statement.

k. Dummy arguments that were defined for a subprogram by some previous reference to the subprogram are undefined for subsequent entry into the subprogram.

l. The value of the function must be returned by using the current entry name.

Table 15-3
DECsystem-20 FORTRAN Library Subroutines

Subroutine Name	Effect
DATE	Places today's date as left-justified ASCII characters into a dimensioned 2-word array. CALL DATE (array) where array is the 2-word array. The date is in the form dd-mmm-yy where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-digit month (e.g., Mar), and yy is a 2-digit year. The data is stored in ASCII code, left-justified, in the two words.
DEFINE FILE	A DEFINE FILE call can be used to establish and define the structure of each file to be used for random access I/O operations. **NOTE** **The OPEN statement may be used to perform the same functions as DEFINE FILE.** The format of a DEFINE FILE call may be CALL DEFINE FILE (u,s,v,f,proj,prog) where
DEFINE FILE (cont)	*u* = logical FORTRAN device numbers. *s* = the size of the records which comprise the file being defined. The argument *s* may be an integer constant or variable. *v* = an associated variable. The associated variable is an integer variable that is set to a value that points to the record that immediately follows the last record transferred. This variable is used by the FIND statement (Chapter 10). At the end of each FIND operation the variable is set to a value that points to the record found. The variable *v* cannot appear in the I/O list of any I/O statement that accesses the file set up by the DEFINE FILE statement. *f* = filename to be given the file being defined.[1] *proj* = user's project number. *prog* = user's programmer's number.

[1]Refer to Appendix B for detailed information on how to specify a directory for the DECsystem-20.

Table 15-3 (Cont)
DECsystem-20 FORTRAN Library Subroutines

Subroutine Name	Effect
	Example The statement CALL DEFINE FILE (1,10,ASCVAR,'FORTFL.DAT',0,0) establishes a file named FORTFL.DAT on device 01 (i.e., disk) which contains word records. The associated variable is ASCVAR, and the file is in the user's area.
DUMP	Causes particular portions of core to be dumped and is referred to in the following form: CALL DUMP (L_1,U_1,F_1,. . .,L_n,U_n,F_n) where L_1 and U_1 are the variable names which give the limits of core memory to be dumped. Either L_1 or U_1 may be upper or lower limits. F_1 is a number indicating the format in which the dump is to be performed: 0 = octal, 1 = real, 2 = integer, and 3 = ASCII. If F is not 0, 1, 2, 3, the dump is in octal. If F_n is missing, the last section is dumped in octal. If U_n and F_n are missing, an octal dump is made from L to the end of the job area. If L_n, U_n, and F_n are missing, the entire job area is dumped in octal. The dump is terminated by a call to EXIT.
ERRSET	Allows the user to control the typeout of execution-time arithmetic error messages, ERRSET is called with one argument in integer mode. CALL ERRSET(N) Typeout of each type of error message is suppressed after N occurrences of that error message. If ERRSET is not called, the default value of N is 2.
EXIT	Returns control to the Monitor and, therefore, terminates the execution of the program.
ILL	Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating point/double precision input, the corresponding word is set to zero.

Table 15-3 (Cont)
DECsystem-20 FORTRAN Library Subroutines

Subroutine Name	Effect
	CALL ILL
LEGAL	Clears the ILLEG flag. If the flag is set and an illegal character is encountered in the floating point/double precision input, the corresponding word is set to zero.
	CALL LEGAL
PDUMP	CALL PDUMP(L_1,U_1,F_1,. . .,L_n,U_n,F_n)
	The arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.
RELEAS	CALL RELEAS(unit*)
	Closes out I/O on a device initialized by the FORTRAN Operating System and returns it to the uninitialized state.
SAVRAN	SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SETABL	CALL SETABL(I,J)
	Specifies a character set where I is an integer which gives the number of the desired character set. If a character set has been defined by I, the value of J is set to 0; if not, J is set to -1. The standard ASCII character set is defined as 1.
SETRAN	SETRAN has one argument which must be a non-negative integer $< 2^{31}$. The starting value of the function RAN is set to one value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.

Table 15-3 (Cont)
DECsystem-20 FORTRAN Library Subroutines

Subroutine Name	Effect
TIME	Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh:mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form bss.t where ss is the seconds, t is the tenths of a second, and b is a blank.

CHAPTER 16
BLOCK DATA SUBPROGRAMS

16.1 INTRODUCTION

Block data subprograms are used to initialize data to be stored in any common areas. Only specification and DATA statements are permitted (i.e., DATA, COMMON, DIMENSION, EQUIVALENCE, and TYPE) in block subprograms. A subprogram of this type must start with a BLOCK DATA statement.

If any entry of a labeled common block is initialized by a BLOCK DATA subprogram, the entire block must be included even though some of the elements of the block do not appear in DATA statements.

Initial values may be entered into more than one labeled common block in a single subprogram of this type.

An executable program may contain more than one block data subprogram.

16.2 BLOCK DATA STATEMENT

The form of the BLOCK DATA statement is

BLOCK DATA name

where

name is a symbolic name given to identify the subprogram.

APPENDIX A
ASCII-1968 CHARACTER CODE SET

The character code set defined in the X3.4-1968 Version of the American National Standard for Information Interchange (ASCII) is given in the following matrix.

1st 2 octal digits	Last octal digit 0	1	2	3	4	5	6	7
00x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
01x	BS	HT	LF	VT	FF	CR	SO	SI
02x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
03x	CAN	EM	SUB	ESC	FS	GS	RS	US
04x	♮	!	"	#	$	%	&	'
05x	(	)	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	\	]	^(↑)	_(←)
14x	grave	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	{	|	}	~(ESC)	DEL

Graphic subsets 64 95

Characters inside parentheses are ASCII-1963 Standard.

NUL	Null
SOH	Start of Heading
STX	Start of Text
ETX	End of Text
EOT	End of Transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal Tabulation
LF	Line Feed
VT	Vertical Tabulation
FF	Form Feed
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1
DC2	Device Control 2
DC3	Device Control 3
DC4	Device Control 4
NAK	Negative Acknowledge
SYN	Synchronous Idle
ETB	End of Transmission Block
CAN	Cancel
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator
DEL	Delete (Rubout)

APPENDIX B
SPECIFYING DIRECTORY AREAS

DECsystem-20 FORTRAN has two ways in which the user can access another user's directory. The first way is via a logical name in place of the device name; the second way is via a project-programmer number instead of a directory name. Either method can be used with FORTRAN; however, use of a logical name is recommended.

NOTE

When the user sees a project-programmer number (i.e., a number similar to [4,204]) in this manual or in an error message, he can use the TRANSL program to find out its corresponding directory name. Refer to Section B.1.1.

For more information about referencing other user's files, refer to the DECsystem-20 USER'S GUIDE.

B.1 USING LOGICAL NAMES

To use a logical name in accessing another user's directory, the user:

1. Gives the DEFINE system command to define a logical name (of no more than six characters) as the other user's directory name.

2. Uses the logical name in place of the device name when typing the file specification.

B.1.1 Giving The DEFINE Command

To give the DEFINE command, the user:

1. Types DEFINE and presses the ESC key; the system prints (LOGICAL NAME).

   @DEFINE (LOGICAL NAME)

2. Types the logical name (ending it with a colon is optional) and presses the ESC key. The system prints (AS).

   @DEFINE (LOGICAL NAME) BAK: (AS)

3. Types the directory name (enclosed in angle brackets) and presses the RETURN key. The system prints an @.

   @DEFINE (LOGICAL NAME) BAK: (AS) <BAKER>

To check the logical name, the user can give the INFORMATION (ABOUT) LOGICAL-NAMES system command.

```
@INFORMATION (ABOUT) LOGICAL–NAMES
BAK = > <BAKER>
@
```

B.1.2 Using The Logical Name

The user can then include the logical name in with FORTRAN by typing the logical name in place of a device name.

The following example shows how the user would output a log file to the directory named <BAKER>. (Remember he has already defined the logical name BAK: as <BAKER>.)

```
@FORTRA
*BAK:TEST
```

B.2 USING PROJECT-PROGRAMMER NUMBERS

To use a project-programmer number in accessing another user's directory, the user:

1. Runs the TRANSL program to find the corresponding project-programmer number for the desired directory name.

2. Includes the project-programmer number after the file type.

The user does not have to define a logical name when he uses a project-programmer number; however, project-programmer numbers may not remain constant over time. Logical names should be used whenever possible.

B.2.1 Running The TRANSL Program

To run the TRANSL program, the user:

1. Types TRANSL and presses the RETURN key. The system prints TRANSLATE (DIRECTORY).

   ```
   @TRANSL
   TRANSLATE (DIRECTORY)
   ```

2. Types the directory name and presses the RETURN key. The system prints the corresponding project-programmer number.

   ```
   @TRANSL
   TRANSLATE (DIRECTORY) BAKER
   <BAKER> IS [4,204]
   @
   ```

The user can also use the TRANSL program to verify that a project-programmer number is correct. He simply replaces the directory name with the project-programmer number.

```
@TRANSL
TRANSLATE (DIRECTORY) [4,204]
[4,204] IS <BAKER>
@
```

B.2.2 Using The Project-Programmer Number

The user can use the project-programmer number with FORTRAN by typing the project-programmer number after the file type.

The following example shows how the user compiles a FORTRAN program from the directory named BAKER, using a project-programmer number. (Remember he has already translated the directory name.)

```
@FORTRA
*TEST.REL,TEST.LST = TEST.FOR [4,204]
```

APPENDIX C
USING THE COMPILER

This appendix explains how to access DECsystem-20 FORTRAN and how to make use of the information it provides. The reader should be familiar with the FORTRAN language and the DECsystem-20 operating system.

C.1 RUNNING THE COMPILER

The command to run FORTRAN is

```
@FORTRA
```

The compiler responds with an asterisk (*) and is then ready to accept a command string. A command is of the general form

```
object filename, listing filename=source filename(s)
```

The following options are given to the user:

1. The user may specify more than one input file in the compilation command string. These files will be logically concatenated by the compiler and treated as one source file.

2. Program units need not be terminated at file boundaries and may consist of more than one file.

3. If no object filename is specified, no relocatable binary file is generated.

4. If no listing filename is specified, no listing is generated.

5. If no type is given, the defaults are .LST (listing), .REL (relocatable binary), and .FOR (source) for their respective files.

C.1.1 Switches Available with DECsystem-20 FORTRAN

Switches to DECsystem-20 FORTRAN are accepted anywhere in the command string. They are totally position and file independent. The switches are shown in Table C-1.

Table C-1
FORTRAN Compiler Switches

Switch	Meaning	Defaults
CROSSREF	Generate a file that can be input to the CREF program.	OFF
DEBUG	See Section C.1.1.1.	OFF
EXPAND	Include the octal-formatted version of the object file in the listing.	OFF
INCLUDE	Compile a D in card column 1 as a space.	OFF
MACROCODE	Add the mnemonics translation of the object code to the listing file.	OFF
NOERRORS	Do not print error messages on the terminal.	OFF
NOWARNINGS	Do not output warning messages.	OFF
OPTIMIZE	Perform global optimization.	OFF
SYNTAX	Perform syntax check only.	OFF

Each switch must be preceded by a slash (/). Switches need consist of only those letters that are required to make the switch unique. But users are encouraged to use at least three letters to prevent conflict with switches in future implementations.

Example

```
@FORTRA
*OFILE,LFILE=SFILE/MAC,S2FILE
```

The /MAC switch will cause the MACRO code equivalent of SFILE and S2FILE to appear in LFILE.LST.

C.1.1.1 The /DEBUG Switch – Using the /DEBUG switch tells FORTRAN to compile a series of debugging features into the user program. Several of these features are specifically designed to be used with FORDDT. Refer to Appendix F for more information. By adding the modifiers listed in Table C-2, the user is able to include specific debugging features.

Table C-2
Modifiers to /DEBUG Switch

Modifiers	Meaning
:DIMENSIONS	Generates dimension information in .REL file for FORDDT.
:TRACE	Generates references to FORDDT required for its trace features (automatically activates :LABELS).
:LABELS	Generates a label for each statement of the form "line-number L." (This option may be used without FORDDT.)
:INDEX	Forces DO LOOP indices to be stored at the beginning of each iteration rather than held in a register for the duration of the loop.
:BOUNDS	Generates the bounds checking code for all array references. Bounds violations will produce run-time error messages. Note that the technique of specifying dimensions of 1 for subroutine arrays will cause bounds check errors. (This option may be used without FORDDT.)

The format of the /DEBUG switch and its modifiers is as follows:

/DEBUG:modifier

or

/DEBUG:(modifier list)

Options available with the /DEBUG modifiers are:

1. No debug features – Either do not specify the /DEBUG switch or include /DEBUG:NONE.

2. All debug features – Either /DEBUG or /DEBUG:ALL.

3. Selected features – Either a series of modified switches; i.e.,

   /DEBUG:BOU /DEBUG:LAB

   or a list of modifiers

   /DEBUG:(BOU,LAB,. . .)

4. Exclusion of features (if the user wishes all but one or two modifiers and does not wish to list them all, he may use the prefix "NO" before the switch he wishes to exclude). The exclusion of one or more features implicitly includes all the others, i.e., /DEBUG:NOBOU is the same as /DEBUG:(DIM,TRA,LAB,IND).

If more than one statement is included on a single line, only the first statement will receive a label (/DEBUG:LABELS) or FORDDT reference (/DEBUG:TRACE). (The /DEBUG option and the /OPTIMIZE option cannot be used at the same time.)

NOTE

If a source file contains line sequence numbers that occur more than once in the same subprogram, the /DEBUG option cannot be used.

The following formulas may be used to determine the increases in program size that will occur due to the addition of various /DEBUG options.

DIMENSIONS:	For each array, have 3+3*N words where N is the number of dimensions, and up to 3 constants for each dimension.
TRACE:	One instruction per executable statement.
LABELS:	No increase.
INDEX:	One instruction per inner loop plus one instruction for some of the references to the index of the loop.
BOUNDS:	For each array, the formula is the same as DIMENSIONS:.
	For each reference to an array element, use 5+N words where N is the number of dimensions in the array. If BOUNDS: was not specified, approximately 1+3*(N-1) words would be used.

C.1.2 LOAD-Class Commands

FORTRAN can also be invoked by using one of the LOAD-class commands. These commands cause the system to interpret the command and construct new command strings for the system program actually processing the command.

COMPILE
LOAD
EXECUTE
DEBUG

Example

```
@EXECUTE(FROM)ROTOR
```

The compiler switches OPT, CREF, and DEBUG may be specified directly in LOAD-class commands and may be used globally or locally.

Example

```
@EXECUTE(FROM)/CREFP1.FOR,P2.FOR/DEBUG:NOBOU
```

The other compiler switches must be passed in parentheses for each specific source file.

Refer to the *DECsystem-20 User's Guide* for further information.

C.2 READING A DECsystem-20 FORTRAN LISTING

When a user requests a listing from the FORTRAN compiler, it contains the following information:

1. A printout of the source program plus an internal sequence number assigned to each line by the compiler. This internal sequence number is referenced in any error or warning messages generated during the compilation. If the input file is line sequenced, the number from the file is used. If code is added via the INCLUDE statement, all INCLUDEd lines will have an asterisk (*) appended to their line sequence number.

2. A summary of the names and relative program locations (in octal) of scalars and arrays in the source program plus compiler generated variables.

3. All COMMON areas and the relative locations (in octal) of the variables in each COMMON area.

4. A listing of all equivalenced variables or arrays and their relative locations.

5. A listing of the subprograms referenced (both user defined and DECsystem-20 FORTRAN defined library functions).

6. A summary of temporary locations generated by the compiler.

7. A heading on each page of the listing containing the program unit name (MAIN., program, subroutine or function, principal entry), the input filename, the list of compiler switches, and the date and time of compilation.

8. If the /MACRO switch was used, a mnemonic printout of the generated code is appended to the listing. This section has four fields:

   LINE: This column contains the internal sequence number of the line corresponding to the mnemonic code. It appears on the first of the code sequence associated with that internal sequence number. An asterisk indicates a compiler inserted line.

   LOC: The relative location in the object program of the instruction.

   LABEL: Any program or compiler generated label. Program labels have the letter "P" appended. Labels generated by the compiler are followed by the letter "M". Labels generated by the compiler and associated with the /DEBUG:LABELS switch consist of the internal sequence number followed by an "L".

   GENERATED CODE: The MACRO mnemonic code.

9. A list of all argument areas generated by the compiler. A zero argument appears first followed by argument blocks for subroutine calls and function references (in order of their appearance in the program). Argument blocks for all I/O operations follow this.

10. Format statement listings.

11. A summary of errors detected or warning messages issued during compilation.

C.2.1 Compiler Generated Variables

In certain situations the compiler will generate internal variables. Knowing what these variables represent can help in reading the macro expansion. The variables are of the form:

.letter digit digit digit digit

i.e., .S0001

where:

Letter	Function of Variable
S	Result of the DO LOOP step size expression of computed iteration count for a loop.
O	Result of a common subexpression or constant computation.
I	Result of a DO LOOP initial value expression or parameters of an adjustably dimensioned array.
F	Arithmetic statement function formal parameters.
R	Result of reduced operator strength expression (D.2.1.2).

The user may find these variables on the listing under SCALARS and ARRAYS.

The following example shows a listing where all these features are pointed out.

Name of Program *Name of Source File* *MACRO code equivalent*

```
MAIN.   TM1.FOR        FORTRAN V.4A(317) /K17M 25-JAN-76        15:51    PAGE 1


00100            IMPLICIT INTEGER (A-Z)
00200            DIMENSION A(100,200),B(100,200)
00300            SUM1=0
00400            SUM2=0
00500            DO 100 J=1,200
00600            DO 100 I=1,100
00700            K1 = I * J
00800            IF(K1 .LT. 500 .OR. K1 .GT. 1500) K1=0
00900            A(I,J)=K1
01000            K2=I+J
01100            IF(K2 .EQ. 100 .OR. K2 .EQ. 200 .OR. K2.EQ.300) K2=K2+1
01200            B(I,J)=K2
01300            SUM1=SUM1+K1
01400            SUM2=SUM2+K2
01500   100      CONTINUE
01600   C
01700            TYPE 10,SUM1,SUM2
01800   10       FORMAT(/H SUM1= ,I9,10H     SUM2= ,I9)
01900            END

SUBPROGRAMS CALLED

                    The relative address of all variables is given.

SCALARS AND ARRAYS ( "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED )

*K1     I       B       2       *J      47042   A       47043   .S0001  116103  .S0000  116104
*SUM2   116105  *I      116106  *K2     116107  *SUM1   116110
```

Compiler generated variables

Internal sequence number on first instruction that goes with that line

octal displacement of instruction.

```
LINE     LOC      LABEL    GENERATED CODE

         0                 JFCL     0,0
         1                 JSP      16,.RESET.
         2                          0,0
300      3                 SETZB    2,SUM1
400      4                 MOVEM    2,SUM2
500      5                 MOVE     2,[777740000001]
         6                 HLREM    2,,-30000
         7        2M:
                           HRRZM    2,J
600      10       3M:
                           MOVE     2,[777763400000011
700      11       4M:
                           MOVE     3,J
         12                IMULI    3,0(2)
         13                MOVEM    3,K1
800      14                CAIL     3,764
         15                CAILE    3,2734
         16                JRST     0,6M
         17                JRST     0,5M
800      20       6M:                                    compiler generated label
                           SETZB    4,K1
900      21       5M:
                           MOVEI    3,144
         22                IMUL     3,J
         23                ADDI     3,0(2)
         24                MOVE     4,K1
         25                MOVEM    4,A-145(3)
1000     26                MOVE     3,J
         27                ADDI     3,0(2)
         30                MOVEM    3,K2
1100     31                MOVE     5,K2
         32                CAIE     5,144
         33                CAIN     5,310
         34                JRST     0,8M
         35       9M:
                           CAIN     5,454
```

```
1100     36        8M:
                             AOS       3.K2
1200     37        7M:
                             MOVEI     3.144
         40                  IMUL      3.J
         41                  ADDI      3.0(2)
         42                  MOVE      5.K2
         43                  MOVEM     5.B-145(3)
1300     44                  ADDM      4.SUM1
1400     45                  ADDM      5.SUM2
1500     46        100P:<----------------------------------- program label
                             AOBJN     2.4M
         47                  AOS       2.J
         50                  AOSGE     0..50000
         51                  JRST      0.3M
1700     52                  MOVEI     16.10M
         53                  PUSHJ     17.OUT.
         54                  MOVEI     16.11M
         55                  PUSHJ     17.IOLST.
1900     56                  MOVEI     16.1M
         57                  PUSHJ     17.EXIT.

C9       ARGUMENT BLOCKS:<----------------------------------- argument blocks
         60                  0..0
         61        1M:       0..0
         62                  777773..0
         63        10M:      0..777777
         64                  0..0
         65                  0..0
         66                  340..100P
         67                  0..7
         70                  0..0
         71        11M:      1100..SUM1
         72                  1100..SUM2
         73                  4000..0
FORMAT STATEMENTS (IN LOW SEGMENT):

1800     116111    10P:      (7H S
         116112              UM1=
         116113              .19.1
         116114              0H
         116115               SUM2
         116116              = .19
         116117              )

MAIN.    ( NO ERRORS DETECTED )
```

C.3 ERROR REPORTING

If an error occurs during the initial pass of the compiler (while the actual source code is being read and processed), an error message is printed on the listing immediately following the line in which the error occurred. Each error references the internal sequence number of the incorrect line. The error messages along with the statement in error are output to the user terminal. For example:

```
@EXECUTE(FROM) DAY.FOR
FORTRAN: DAY
01300                          K1
?FTNNRC LINE:01300             STATEMENT NOT RECOGNIZED
01500  100                     CONTINE
?FTNMSP LINE:01500             STATEMENT NAME MISSPELLED
01600  ?
?FTNICL LINE:01600             ILLEGAL CHARACTER C IN LABEL FIELD

?FTNFTL  MAIN.                 3 FATAL ERRORS AND NO WARNINGS
LINK:  LOADING
[LNKNSA NO START ADDRESS]

EXIT

@
```

If errors are detected after the initial pass of the compiler, they appear in the list file after the end of the source listing. They are output to the user terminal without the statement in error but they do reference its internal sequence number.

C.3.1 Fatal Errors and Warning Messages

There are two levels of messages, warning and fatal error. Warning messages are preceded by "%" and indicate a possible problem. The compilation will continue, and the object program will probably be correct. Fatal errors are preceded by a "?". If a fatal error is encountered in any pass of the compiler, the remaining passes will not be called. Additional errors that would be detected in later compiler passes may not become apparent until the first errors are corrected. As the word fatal denotes, it is not possible to generate a correct object program for a source program containing a fatal error.

The format of messages is

```
?FTNXXX LINE:n text
```

or

```
%FTNXXX LINE:n text
```

where:

?	=	fatal
%	=	warning
FTN	=	FORTRAN mnemonic
XXX	=	3-letter mnemonic for the error message
LINE:n	=	line number where error occurred
text	=	explanation of error

The printing of fatal errors and warning messages on the user's terminal can be suppressed by the use of the /NOERRORS switch; however, messages will still appear on the listing. The /NOWARNINGS switch will suppress warning messages on both user terminal and listing.

C.3.2 Message Summary

At the end of the listing file and on the terminal, a message summary is printed after each program unit is compiled. This message has two forms:

1. when one or more messages were issued

   { ?FTNFTL / %FTNWRN } name NO/number FATAL ERRORS AND NO/number WARNINGS

   or

2. when no messages were issued

   name [NO ERRORS DETECTED]

where name is the program or subprogram name. ([NO ERRORS DETECTED] appears on the listing only.) For a complete list of fatal errors and warning messages, see Appendix G.

C.4 CREATING A REENTRANT FORTRAN PROGRAM WITH LINK

To produce a sharable program, load the object file into memory and give the SAVE command as follows:

```
@LOAD (FROM) MAIN.REL
LINK:loading
@SAVE
MAIN.EXEI SAVED
@
```

Users can then run the program using the run command

```
@RUN MAIN
```

APPENDIX D
WRITING USER PROGRAMS

This appendix is a guide for writing effective user programs with FORTRAN. It contains techniques for optimization, interaction with non-FORTRAN programs, and other useful programming hints.

D.1 GENERAL PROGRAMMING CONSIDERATIONS

Programming considerations that should be observed when preparing a FORTRAN program to be compiled by DECsystem-20 FORTRAN are described in the following paragraphs.

D.1.1 Accuracy and Range of Double Precision Numbers

Floating point and real numbers may consist of up to 16 digits in a double precision mode. Their range is specified in Chapter 3, Section 3.2 of this manual. Care must be taken when testing the value of a number within the specified range since, although numbers up to 10^{38} may be represented, DECsystem-20 FORTRAN can only test numbers of up to eight significant digits (REAL precision) and 16 significant digits (DOUBLE precision).

Care must also be taken when testing the floating point computation for a result of 0. In most cases the anticipated result (i.e., 0) will be obtained; however, in some cases the result may be a very small number that approximates 0. Such an approximation of 0 would cause tests within statements (i.e., an arithmetic IF) to fail.

D.1.2 Writing FORTRAN Programs for Execution on Non-DEC Machines

If a program is to be prepared to run on both a DECsystem-20 computer and a non-DEC machine, the user should

1. avoid using the non-ANSI Standard features of DECsystem-20 FORTRAN, and

2. consider the accuracy and size of the numbers that the non-DEC machine is capable of handling.

D.1.3 Using Floating Point DO Loops

DECsystem-20 FORTRAN permits the user to employ non-integer single or double precision numbers as the parameter variables in a DO statement. The primary advantage of the foregoing is to enable the user to generate a wider range of values for the DO loop index variables which may, in turn, be used inside the loop for computations. Care should be taken in considering the loss of precision that may occur in this context.

D.1.4 Computation of DO Loop Iterations

The number of times through a DO loop is computed outside the loop and is not affected by any changes to the DO index parameters within the loop. The formula for the number of times a DO loop is executed is

DO 10 I=m_1, m_2, m_3

$$\text{MAX}\left(1, \frac{m_2 - m_1 + m_3}{m_3}\right) = \text{Number of cycles}$$

The values of the parameters (i.e., m_1, m_2, m_3) may be of any type; however, proper consideration must be given to the foregoing formula, particularly when using logicals. One iteration of each DO loop is always performed even if the result of the foregoing calculation is less than or equal to zero.

D.1.5 Subroutines – Programming Considerations

The following items must be considered when preparing and executing subroutines:

1. During execution, no check is made to see if the proper number of parameters were passed.

2. If the number of actual arguments passed to a subroutine are less than the number of dummy arguments specified, the values of the unspecified arguments are undefined.

3. If the number of actual arguments passed to a subroutine is greater than the number of dummy arguments given, the excess arguments are ignored.

4. If an actual parameter is a constant and its corresponding dummy argument is set to another value, all references made to the constant in the calling program may be changed to the value of the dummy argument.

5. No check is made to see if the parameters passed are of the same type as the dummy parameters. If an actual parameter is a constant and the corresponding dummy is of type real, be sure to include the decimal point with the constant. If the dummy is double precision, be sure to specify the constant with a "D".

   Examples

   If the function F(A) is called by inputting F(2) and A is type real, F interprets the integer 2 as an unnormalized floating point number. In this instance, F(A) should be called with F(2.0).

   Similarly, if the function F1(D) is called by inputting F1(2.5) and D is double precision, F1 assumes that its parameters have been specified with two words of precision and picks up whatever follows the constant 2.5 in core. The proper method is to use F1(2.5D00).

NOTE

No notice is given to the user if any of the situations described in items 1, 2, 3, 4, and 5 occur.

D.1.6 Reordering of Computations

Computations that are not enclosed within parentheses may be reordered by the compiler. Sometimes it is necessary to use parentheses to prevent improper results from being obtained from a specific computation.

For example, assuming that

1. RL1 represents a large number such that RL1*RL2 will cause an overflow condition, and

2. RS1 is a very small number (i.e., less than 1) the program sequence

```
        .
        .
        .
A = RS1*RL1*RL2
B = RS2*RL2*RL1
        .
        .
        .
```

will not produce an overflow when evaluated left to right since the first computation in each expression (i.e., RS1*RL1 and RS2*RL2) will produce an interim result that is smaller than either large number (RL1 or RL2).

However, the compiler will recognize RL1*RL2 as a common subexpression and generate the following sequence:

```
temp =  RL1*RL2
A    =  RS1*temp
B    =  RS2*temp
```

The computation of temp will cause an overflow.

The program sequence should be written in the following manner to ensure that the desired results are obtained:

```
        .
        .
A = (RS1*RL1)*RL2
B = (RS2*RL2)*RL1
```

Computations may be reordered even when global optimization is not selected.

D.1.7 Dimensioning of Formal Arrays

When an array is specified as a formal parameter to a subprogram unit, it is necessary to indicate to the compiler that the parameter is an array. The user must dimension the array in a specification statement. This is the only way the compiler is able to distinguish a reference to such an array from a function reference. Designating the array with a dimension of 1 has become a common practice among users.

Example

```
SUBROUTINE SUB1(A,B)
DIMENSION A(1)
```

There are disadvantages to using the above technique because the dimension information provided is not adequate in some cases, specifically

1. Reading or writing the array by name.

```
DIMENSION ARRAY (10)
READ (1) ARRAY
```

This is a binary read that will read 10 words into ARRAY.

```
SUBROUTINE SUB1(A)
DIMENSION A(1)
READ(1)A
```

This binary read will cause 1 word to be read into A.

2. Reading the array as a format

```
SUBROUTINE SUB2 (FMT)
DIMENSION FMT(1)
READ (1,FMT)
```

This will cause 1 word of the array FMT to be written over with the characters read from the record on unit 1.

When using the /DEBUG:BOUNDS compilation switch, the dimension information used is that which is specified in the array declaration.

```
SUBROUTINE DO IT(A)
DIMENSION A(1)
A(2) = 0
```

The reference to A(2) will cause the out-of-bounds warning message to be generated.

D.2 DECsystem-20 FORTRAN GLOBAL OPTIMIZATION

The user has the option of invoking a global optimizer during compilation. This optimizer treats groups of statements in the source program as a single entity. The purpose of the global optimizer is to prepare a more efficient object program that produces the same results as the original unoptimized program but takes significantly less execution time. The output of the lexical and syntax analysis phase of the compiler is developed into an optimized source program equivalent (in results) to the original. The optimized program is then processed by the standard compiler code generation phase.

D.2.1 Optimization Techniques

D.2.1.1 Elimination of Redundant Computations – Often the same subexpression will appear in more than one computation throughout a program. If the values of the operands of such a common expression are not changed between computations, the subexpression may be written as a separate arithmetic expression, and the variable

representing its resultant may then be substituted where the subexpression appears. This eliminates unnecessary recomputation of the subexpression. For example, the instruction sequence

```
A = B*C+E*F
      .
      .
      .
H = A+G-B*C
      .
      .
      .
IF((B*C)-H) 10,20,30
```

contains the subexpression B*C three times when it really needs to be computed only once. Rewriting the foregoing sequence as

```
T = B*C
A = T+E*F
      .
      .
H = A+G-T
      .
      .
IF((T)-H) 10,20,30
```

eliminates two computations of the subexpression B*C from the overall sequence.

Decreasing the number of arithmetic operations performed in a source program by the elimination of common subexpressions shortens the execution time of the resulting object program.

D.2.1.2 Reduction of Operator Strength – The time required to execute arithmetic operations will vary according to the operator(s) involved. The hierarchy of arithmetic operations according to the amount of execution time required is

MOST TIME	OPERATOR
↓	**
	/
	*
LEAST TIME	+,-

During program optimization, the global optimizer replaces, where possible,[1] some arithmetic operations that require the most time with operations that require less time. For example, consider the following DO loop that is used to create a table for the conversion of from 1 to 20 miles to their equivalents in feet.

```
      DO 10 MILES = 1,20
10    IFEET(MILES) = 5280*MILES
```

[1] Numerical analysis considerations severely limit the number of cases where it is possible.

The execution time of the foregoing loop would be shorter if the time consuming multiply operation (i.e., 5280*MILES) could be replaced by a faster operation. Since the variable MILES is incremented by 1 on each iteration of the loop, the multiply operation may be replaced by an add and total operation.

In its optimized form, the foregoing loop would be replaced by a sequence equivalent to

```
      K=5280
      DO 10 MILES = 1,20
      IFEET(MILES) = K
10    K=K+5280
```

In the optimized form of the loop, the value of K is set to 5280 for the first iteration of the loop and is increased by 5280 for each succeeding iteration of the loop.

The foregoing situation occurs frequently in subscript calculations which implicitly contain multiplications whenever the dimensionality is two or greater.

D.2.1.3 Removal of Constant Computation From Loops – The speed with which a given algorithm may be executed can be increased if instructions and/or computations are moved out of frequently traversed program sequences into less frequently traversed program sequences. Movement of code is possible only if none of the arguments in the items to be moved are redefined within the code sequences from which they are to be taken. Computations within a loop comprised of variables or constants that are not changed in value within the loop may be moved outside the loop. Decreasing the number of computations made within a loop will greatly decrease the execution time required by the loop.

For example, in the sequence

```
      DO 10 I=1,100
10    F=2.0*Q*A(I)+F
```

the value of the computation 2.0*Q, once calculated on the first iterations, will remain unchanged during the remaining 99 iterations of the loop. Reforming the foregoing sequence to:

```
      QQ = 2.0*Q
      DO 10 I=1,100
10    F=QQ*A(I)+F
```

moves the calculation 2.0*Q outside of the scope of the loop. This movement of code eliminates 99 multiply operations.

In addition it is possible to remove entire assignment statements from loops. This action can be easily detected from the macro expanded listings. The internal sequence number remains with the statement and appears out of order in the leftmost column of the macro expanded listing (LINE).

D.2.1.4 Constant Folding and Propagation – In this method of optimization, expressions containing determinate constant values are detected and the constants are replaced, at compile time, by their defined or calculated value. For example, assume that the constant PI is defined and used in the following manner:

```
      .
      .
      .
PI = 3.14159
      .
      .
      .
X = 2*PI*Y
      .
      .
      .
```

At compile time, the optimizer will have used the defined value of PI to calculate the value of the subexpression 2*PI. The optimized sequence would then be

```
      .
      .
PI = 3.14159
      .
      .
      .
X = 6.28318*Y
      .
      .
```

thereby eliminating a multiply operation from the object code program.

The computation of determinate constant values at compile time is termed "folding"; the use of the defined value of a constant for replacement purposes throughout a program sequence is termed "propagation of the constants". The execution time saved by the foregoing type of compile time optimization is particularly important when the modified instruction occurs in a loop.

D.2.1.5 Removal of Inaccessible Code – The optimizer detects and eliminates any code within the source program that cannot be accessed. In general, the foregoing condition will not exist since programmers will not normally include such code in their programs; however, inaccessible code may appear in a program during the debugging process. The removal of inaccessible code by the optimizer will reduce the size of the optimized object program. A warning message is generated for each inaccessible line removed.

D.2.1.6 Global Register Allocation – During the compilation of a source program the optimizer controls the allocation of registers to minimize computation time in the optimized object program. The intent of the allocation process is to minimize the number of MOVE and MOVEM machine instructions that will appear in the most frequently executed portions of the code.

D.2.1.7 I/O Optimization – Every effort is made to minimize the number of calls required into the FOROTS system. This is done primarily through extensive analysis of implied DO loop constructs on READ, WRITE, ENCODE, DECODE and REREAD statements. The formats of these special blocks are described in Appendix E. These optimizations reduce the size of the program (argument code plus argument block size is reduced) and greatly improve the performance of programs that use implied DO loop I/O statements.

D.2.1.8 Uninitialized Variable Detection – A warning message is generated when a scalar variable is referenced before it could possibly have received a value.

D.2.1.9 Test Replacement – If the only use of a DO loop index is to reduce operator strength (D.2.1.2) and the loop does not contain exits (GO TO's out of the loop), the DO loop index is not needed and can be replaced by the reduced variable. This actually occurs quite often in double precision array subscript computations.

For example:

```
      DO 10 I=1,10
      K=K+7*I
10    CONTINUE
```

Reduction of operator strength and test replacement together transform this loop into

```
      DO 10 I=7,70,7
      K=K+I
10    CONTINUE
```

Although this particular example is trivial, the actual situation occurs frequently in subscript computation.

D.2.2 Improper Function References

The ANSI FORTRAN standard prohibits the use of a function's reference that has side effects that will influence the statement in which the function is referenced (such as defining or redefining other elements in the statement). The compiler depends on strict adherence to the foregoing rule. The generated object program may not yield the desired results if this rule is violated.

D.2.3 Programming Techniques for Effective Optimization

The following recommendations, when observed during the coding of a FORTRAN source program, improve the effectiveness of the optimizer:

1. DO loops with an extended range should not be used.
2. Specify label lists when using assigned GO TO's.
3. Nest loops so that the innermost index is the one with the largest range of values.
4. Avoid the use of associated input/output variables.
5. Avoid unnecessary use of COMMON and EQUIVALENCE.

D.3 INTERACTING WITH NON-DECsystem-20 FORTRAN PROGRAMS AND FILES

WARNING

FOROTS assumes it has complete control of the object time environment. Executing monitor calls in a MACRO subroutine may produce unexpected results. The following guidelines must be observed:

1. **Do not manipulate any file FOROTS has OPEN. This includes implicit manipulation by such calls as RESET, CLOSF, CLZFF, RLJFN etc.**
2. **Do not change the state of the software interrupt system. Do not use the following monitor calls SIR, EIR, DIR, AIC, IIC, DIC, SIRCM, DEBRK, ATI, DTI, and CIS.**
3. **Do not generate any software interrupts.**
4. **Do not attempt to create processes.**

D.3.1 Calling Sequences

The standard procedures for writing subroutine calls for the DECsystem-20 are described in the following paragraphs.

1. Procedure

   a. The calling program must load the right half of accumulator (AC) 16 with the address of the first argument in the argument list.

   b. The left half of AC 16 must be set to zero.

   c. The subroutine is then called by a PUSHJ instruction to AC 17.

   d. The returns will be made to the instruction immediately after the PUSHJ 17 instruction.

   e. If the /DEBUG:BOUNDS option of the FOROTS trace facility is being used, the calling sequence must be

```
MOVEI 16,AP
PUSHJ 17,F
```

   where AP is the pointer to the argument list. If the trace facility is to be used, the word preceding the first word of an entry point should have its name in sixbit.

2. Restrictions

   a. Skip returns are not permitted.

   b. The contents of the pushdown stack located before the address specified by AC 17 belongs to the calling program; it cannot be read by the called subprogram.

   c. FOROTS assumes that it has control of the stack; therefore, the user must not create his own stack. The FOROTS stack is initialized by

```
JSP 16,RESET.
```

   or the library routine

```
CALL RESET.
```

D.3.2 Accumulator Usage

The specific functions performed by accumulators (AC) 17,16,0 and 1 are as follows:

1. Pushdown Pointer – AC 17 is always maintained as a pushdown pointer. Its right half points to the last location in use on the stack and its left half contains the negative of the number of (words-1) allocated to the unused remainder of the stack (a trap occurs when something is pushed into the next to last location. A positive left half is not permitted.

2. Argument List Pointer – AC 16 is used as the argument pointer. The called subprogram does not need to preserve its contents. The calling program cannot depend on getting back the address of the argument list passed to the callee. AC 16 cannot point to the AC's or to the stack.

3. Temporary and Value Return Registers – AC 0 and 1 are used as temporary registers and for returning values. The called subprogram does not need to preserve the contents of AC 0 or 1 (even if not returning a value). The calling program must never depend on getting back the original contents of the data passed to the called subprogram.

4. Returning Values – At the option of the designer of a called subprogram, a subroutine may pass back results by modifying the arguments, returning a single precision value in AC 0 or a double precision or complex value in AC 0 and 1. A combination of the above may be used. However, two single precision values cannot be returned in AC 0 and 1 since FORTRAN would not be able to handle it.

5. Preserved AC's – DECsystem-20 FORTRAN FUNCTION subprograms preserve AC's 2–15; subroutine subprograms do not.

   The design of the called subprogram cannot depend on the contents of any of the AC's being set up by the calling subprogram, except for AC's 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either DECsystem-20 FORTRAN or COBOL.

D.3.3 Argument Lists

The format of the argument list is as follows:

```
                    arg count word
arg list addr. - - - first arg entry
                    second arg entry
                          .
                          .
                          .
                    last arg entry
```

The format of the arg count word is:

bits 0–17	These contain -n, where n is the number of arg entries.
bits 18–35	These are reserved and must be 0.

The format of an arg entry is as follows (each entry is a single word):

bits 0–8	Reserved for future DEC development (set to 0 for now).
bits 9–12	Arg type code.
bit 13	Indirect bit if desired.
bits 14–17	Index field, must be 0 for present.
bits 18–35	Address of the argument.

The following restrictions should be observed:

1. Neither the argument lists nor the arguments themselves can be on the stack. This restriction is imposed so that the stack can be moved. The same restriction applies to any indirect argument pointers.

2. The called program may not modify the argument list itself. The argument list may be in a write-protected segment.

   Note that the arg count word is at position -1 with respect to the contents of AC16. This word is always required even if the subroutine does not handle a variable number of arguments. A subroutine that has no arguments must still provide an argument list consisting of two words (i.e., the argument count word with a 0 in it and a zero argument word).

Example

```
        MOVEI 16, AP        ;SET UP ARG POINTER
        PUSHJ 17,SUB        ;CALL SUBROUTINE
        . . .               ;RETURN HERE
              .
              .
              .
        ;ARGUMENT LIST
        -3,,0
AP:     A
        B
        C

;SUBROUTINE TO SET THIRD ARG TO SUM OF FIRST TWO ARGS

SUB;    MOVE    T,@0(16)    ;GET FIRST ARG
        ADD     T,@1(16)    ;ADD SECOND ARG
        MOVEM   T,@2(16)    ;SET THIRD ARG
        POPJ    17,         ;RETURN TO CALLER
```

D.3.4 Argument Types

Table D-1
Argument Types and Type Codes

Type Code	Description	
	FORTRAN Use	COBOL Use
0	Unspecified	Unspecified
1	FORTRAN Logical	Not applicable
2	Integer	1-word COMP
3	Reserved	Reserved
4	Real	COMP-1
5	Reserved	Reserved
6	Reserved	Reserved
7	Label	Procedure address
10	Double real	Not applicable
11	Not applicable	2-word COMP
12	Reserved	Reserved
13	Reserved	Reserved
14	Complex	Not applicable
15	Not applicable	Byte string descriptor
16	Reserved	Reserved
17	ASCIZ string	Not applicable

Literal arguments are permitted, but they must reside in a writable segment. This is because the FORTRAN compiler makes a local of all non-array elements and copies all formals back to the caller's arguments. All unused type codes are reserved for future Digital development.

D.3.5 Description of Arguments

The types of the arguments that may be passed are:

1. Type 0 – Unspecified

   The calling program has not specified the type. The called subprogram should assume that the argument is of the correct type if it is checking types. If several types are possible, the called subprogram should assume a default as part of its specification. If none of the above conditions are true, the called subprogram should handle the argument as an integer (type 2).

2. Type 1 – FORTRAN logical

   A 36-bit binary value containing 0 or positive to specify .FALSE. and negative to specify .TRUE..

3. Type 2 – Integer and 1-word-COMP

   A 36-bit 2's complement signed binary integer.

4. Type 4 – Real and COMP-1

   A 36-bit DECsystem-10 format floating point number.

bit 0	sign
bits 1–8	excess 128 exponent
bits 9–35	mantissa

5. Type 6 – Label and procedure address

   A 36-bit unsigned binary value.

6. Type 7 – Label and procedure address

   A 23-bit memory address.

bits 0–12	always 0
bit 13	indirect flag
bits 14–17	0
bits 18–35	the address

7. Type 10 – Double real

   A double precision floating point number.

8. Type 11 – 2-word COMP

   A 2-word (72-bit) 2's complement signed binary integer.

word 1, bit 0	sign
word 1, bits 1–35	high order
word 2, bit 0	same as word 1, bit 0
word 2, bits 1–35	low order

9. Type 12 – Double octal

   A 72-bit unsigned binary value.

10. Type 14 – Complex

    A complex number represented as an ordered pair of 36-bit floating point numbers. The first of which represents the real part and the second of which represents the imaginary part.

11. Type 15 – Byte String Descriptor

    The format of the byte string descriptor is

word 1:	ILDB-type pointer (i.e., aimed at the byte preceding the first byte of the string)
word 2:	EXP byte count

    The byte descriptor may not be modified by the called program. The byte string itself must consist of a string of contiguous bytes of a uniform size. The byte size may be any number of bits from 1 to 36. The byte count must be large enough to encompass 256K words of storage, i.e., 24 bits for 1-bit bytes. (Refer to the COBOL Reference Manual.)

12. Type 17 – ASCIZ string

    A string of contiguous 7-bit ASCII bytes left justified on the word boundary of the first word and terminated by a null byte in the last word. The length of the string may be from 1 to 256K words.

D.3.6 Converting Existing MACRO Libraries for Use with DECsystem-20 FORTRAN

The following simple example illustrates the FORTRAN calling sequence.

```
MAIN.   EX1     FORTRAN V.4A(317) /K1/M 26-JAN-76        11:09    PAGE 1


00100   C       AM EXAMPLE OF A CALL TO A SUBROUTINE WITH A VARIETY OF ARGUMENTS
00200
00300           DOUBLE PRECISION DP
00400           DIMENSION B(10)
00500   C       THE ARGUMENTS ARE:
00600   C       1. REAL VARIABLE
00700   C       2. ARRAY NAME
00800   C       3. ARRAY ELEMENT REFERENCE
00900   C       4. INTEGER ELEMENT
01000   C       5. DOUBLE PRECISION VARIABLE
01100   C       6. OCTAL CONSTANT
01200   C       7. LITERAL
01300
01400           CALL SUB1(A,B,B(I),K,DP,"777,'ABC')
01500
01600           END


SUBPROGRAMS CALLED

SUB1

SCALARS AND ARRAYS ( "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED )

 DP      1       *K      3       6       4       *A      16      *I      17

TEMPORARIES

 .00000 20
```

```
LINE     LOC      LABEL    GENERATED CODE

         0                 JFCL     0,0
         1                 JSP      16,RESET.
         2                          0,0
1400     3                 MOVE     2,I
         4                 MOVEI    2,B-1(2)
         5                 MOVEM    2,,00000
         6                 MOVEI    16,2M
         7                 PUSHJ    17,SUB1
*        10                MOVEI    16,1M
         11                PUSHJ    17,EXIT.

 ARGUMENT BLOCKS:

         12                0,,0
         13       1M:      0,,0
         14                777771,,0
         15       2M:      200,,A
         16                200,,B
         17                220,,,00000
         20                100,,K
         21                400,,DP
         22                0,,[000000000777]
         23                740,,[4060503201001
MAIN.    [ NO ERRORS DETECTED ]
```

```
00100           SUBROUTINE SUB1(REAL1,ARYNAM,ARYELM,INT1,DBLPRC,OCT,LIT)
00200           DOUBLE PRECISION DBLPRC
00300           DIMENSION ARYNAM(10)
00400     C
00500     C     THIS SUBROUTINE ILLUSTRATES THE USE AND MODIFICATION OF
00600     C     SOME OF THE ARGUMENT TYPES
00700     C
00800           REAL1=ARYELM
00900           O=ARYNAM(INT1)
01000           OCT="777
01100           RETURN
01200           END
```

SUBPROGRAMS CALLED

SCALARS AND ARRAYS ["*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED]

```
%LIT            *OCT    1       *O      2       *ARYELM 3       %DBLPRC         *REAL1  4
*INT1   5        ARYNAM 6
```

TEMPORARIES

```
LINE     LOC      LABEL    GENERATED CODE

         0                 636542,,210000

         SUB1:
100      0                 MOVE     0.@0(16)
         1                 MOVEM    0.REAL1
         2                 MOVEI    0.@1(16)
         3                 MOVEM    0.ARYNAM
         4                 MOVE     0.@2(16)
         5                 MOVEM    0.ARYELM
         6                 MOVE     1.@3(16)
         7                 MOVEM    1.INT1
         10                MOVE     2.@5(16)
         11                MOVEM    2.OCT
800      12       3M:
                           MOVEM    0.REAL1
900      13                MOVE     2.INT1
         14                ADD      2.ARYNAM
         15                MOVE     2.777777(2)
         16                MOVEM    2.0
1000     17                MOVEI    2.777
         20                MOVEM    2.OCT
1200     21       2M:
                           MOVE     0.REAL1
         22                MOVEM    0.@0(16)
         23                MOVE     0.OCT
         24                MOVEM    0.@5(16)
         25                POPJ     17,0

 ARGUMENT BLOCKS:

         26                0,,0
         27       1M:      0,,0
SUB1     [ NO ERRORS DETECTED ]
```

D.3.7 Interaction with COBOL

The FORTRAN programmer may call COBOL programs as subprograms and, conversely, the COBOL programmers may call FORTRAN programs as subprograms.

In either of the foregoing cases, I/O operation must not be performed in the called subprogram.

D.3.7.1 Calling FORTRAN Subprograms from COBOL Programs – COBOL programmers may write subprograms in FORTRAN to utilize the conveniences and facilities provided by this language. The COBOL verb ENTER is used to call FORTRAN subroutines. The form of ENTER is as follows:

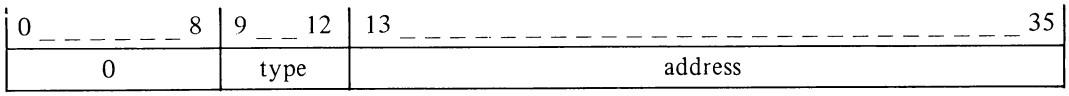
ENTER FORTRAN program name [USING { identifier-1 / literal-1 / procedure name-1 } [[, { identifier-2 / literal-2 / procedure-2 }] ...]

The USING clause of the foregoing forms moves the data within the COBOL program which is to be passed to the called FORTRAN subprogram. The passed data must be in a form acceptable to FORTRAN.

The calling sequence used by COBOL in calling a FORTRAN subprogram is:

```
MOVEI 16, address of first entry in argument list
PUSHJ 17, subprogram address
```

If the USING clause appears in the ENTER statement, an argument list is created which contains an entry for each identifier or literal in the order of appearance in the USING clause. It is preceded by a word containing, in its left half, the negative number of the number of entries in the list. If no USING clause is present, the argument list contains an empty word and the preceding word is set to 0. Each entry in the list is one 36-bit word at the form:



Bits 0–8 are always 0.

Bits 9–12 contain a type code that indicates the USAGE of the argument.

Bits 13–35 contain the address of the argument or the first word of the argument; the address can be indexed or indirect.

The types, their codes, how the codes appear in the argument list, and the locations specified by the addresses are described as follows:

a. For 1-word COMPUTATIONAL items

CODE:	2
IN ARGUMENT LIST:	XWD 100, address
ADDRESS:	that of the argument itself

b. For 2-word COMPUTATIONAL items

CODE:	11
IN ARGUMENT LIST:	XWD 440, address
ADDRESS:	that of the high-order word of the argument

c. For COMPUTATIONAL-1 items

CODE:	4
IN ARGUMENT LIST:	XWD 200, address
ADDRESS:	that of the argument itself

d. For DISPLAY-6 and DISPLAY-7 items

CODE:	15
IN ARGUMENT LIST:	XWD 640, address
ADDRESS:	that of a 2-word descriptor for the argument
WORD1:	a byte pointer to the identifier or literal
WORD2:	bit 0 is 1 if the item is numeric
	bit 1 is 1 if the item is signed
	bit 2 is 1 if the item is a figurative constant (including ALL)
	bit 3 is 1 if the item is a literal
	bits 4 through 11 are reserved for expansion
	bit 12 is 1 if the item has a PICTURE with one or more Ps just before the decimal point (e.g., 99PPV)
	bits 13 through 17 are the number of decimal places. If bit 12 is 1, this is the number of Ps.
	bits 18 through 35 contain the size of the item in bytes.

e. For procedure names (which cannot be used for calls to COBOL subprograms)

CODE:	7
IN ARGUMENT LIST:	XWD 340, address
ADDRESS:	that of the procedure

The return from a subprogram is POPJ17 statement after call.

D.3.7.2 Calling COBOL Subroutines from FORTRAN Programs – To call COBOL subroutines use the standard subroutine calling Mechanism:

CALL COBOLS (args. . .)	subroutine call
X=COBOLS (args. . .)	function call

The COBOL subroutine must have been compiled using the COBOL compiler described in the *COBOL Reference Manual.*

APPENDIX E
FOROTS

This appendix describes the facilities FOROTS provides for the FORTRAN user. FOROTS implements all standard FORTRAN I/O operations as set forth in the "American National Standard FORTRAN, ANSI X3.9-1966." In addition it provides the user with capabilities and programming features beyond those defined in the ANSI standard.

The primarv function of FOROTS is to act as a direct interface between user object programs and the DECsystem-20 monitor during input and output operations. Other capabilities include

1. Job initialization
2. Channel and memory management
3. Error handling and reporting
4. File management
5. Formatting of data
6. Mathematical library
7. User library (non-mathematical)
8. Specialized applications packages
9. Overlay facilities

E.1 FEATURES OF FOROTS

Many specific user features are described briefly in the following list; more detailed information concerning the implementation of these features is given later in this appendix.

1. A user program may run in either batch or timesharing mode without changing the program. All differences between batch mode and timesharing mode operations are resolved by FOROTS.

2. User programs may access both directory and non-directory devices in the same manner.

3. FOROTS helps provide complete data file compatibility between all DECsystem-20 devices.

4. FOROTS does not require line-blocking (a requirement that each output buffer must contain only an integral number of lines).

5. Up to 15 data files may be accessed simultaneously. Any number or all of the open data files may be accessed randomly.

6. FOROTS treats devices located at remote stations similarly to local devices.

7. Programs written for magnetic tape operations will run correctly on disk under FOROTS supervision. FOROTS simulates the commands needed for magnetic tape operations.

8. Object program device and file specifications may be changed or specified via a FOROTS interactive dialogue mode.

9. Non-FORTRAN binary data files may be read in image mode by FOROTS.

10. FOROTS provides interactive program/operating system error processing routines. These routines permit the user to route the execution of the program to specific error processing routines whenever designated types of errors are detected.

11. An error traceback facility for fatal errors provides a history of all subprogram calls made back to the main program together with the address of the point at which the error occurred.

12. FOROTS provides a trap handling system for arithmetic functions, including default values and error reports.

13. ASCII and binary records may be mixed in the same file and both may be accessed in either sequential or random access mode.

14. FOROTS permits the user program to switch from READ to WRITE on the same I/O device without loss of data or buffering.

15. Although primarily designed for use with the DECsystem-20 FORTRAN compiler, FOROTS may also be used as an independent I/O system. FOROTS may be used as an I/O system for MACRO object programs as well as for FORTRAN.

E.2 ERROR PROCESSING

Whenever a run-time error is detected, the FOROTS error processing system takes control of program execution. This system determines the class of the error and either outputs an appropriate message at the controlling user terminal or branches the program to a predesignated processing routine.

E.3 INPUT/OUTPUT FACILITIES

I/O data channel and access modes are individually described in the following paragraphs.

E.3.1 Input/Output Channels Used Internally by FOROTS

Fifteen software channels (1–15) are available in I/O operations. Software channel 0 is reserved for the following system functions:

1. The printing of error messages, and
2. The loading and initialization of FOROTS

Software channels 1 through 15 are available for user program data transfer operations. When a request is made for a data channel, a table is scanned until a free channel is found. The first free channel is assigned to the requesting program; on completion of the assigned transfer, control of the software channel is returned to FOROTS.

E.3.2 File Access Modes

Data may be transferred between processor storage and peripheral devices in two major modes – sequential, and random.

E.3.2.1 Sequential Transfer Mode – In sequential data transfer operations, the records involved are transferred in the same order as they appear in the source file. Each I/O statement executed in this mode transfers the record immediately following the last record transferred from the accessed source file. A special version of the sequential

mode (referred to as Append) is available for output (write) operations. The special Append mode permits the user to write a record immediately after the last logical record of the accessed file. During the Append operation, the records already in the accessed file remain unchanged; the only function performed is the appending of the transferred records to the end of the file.

Transfer modes (other than SEQINOUT) must be specified by setting the ACCESS option of a FORTRAN OPEN statement to one of several possible arguments. For the sequential mode, the arguments are

ACCESS = 'SEQIN' (sequential read-only mode)
ACCESS = 'SEQOUT' (sequential write-only mode)
ACCESS = 'SEQINOUT' (sequential read followed by a sequential write)
ACCESS = 'APPEND' (sequential Append mode)

E.3.2.2 Random Access Mode – This transfer mode permits records to be accessed and transferred from a source file in any desired order. Random access transfers must be made between processor memory and a device that permits random addressing operations (i.e., disk) to files that have been set up for random access. Files for random access must contain a specified number of identically-sized records which may be individually accessed by a record number.

Random access transfers may be carried out in either a read/write mode or a special read-only mode. Random transfer modes must be specified by setting the ACCESS option of an OPEN statement to one of several possible arguments.

ACCESS = 'RANDOM' (random read/write mode)
ACCESS = 'RANDIN' (random special read-only mode)

E.4 ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS

The types of data files that are acceptable to FOROTS are individually described in the following paragraphs.

E.4.1 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters; each set is terminated by a vertical paper-motion character (i.e., Form Feed, Vertical Tab, or Line Feed). All ASCII records start on a word boundary; the last word in a record is padded with nulls, if necessary, to ensure that the record also ends on a word boundary. Logical records may be split across 128-word blocks. There is no implied maximum length for logical records.

NOTE

On sequential input, FOROTS does not require conformation to word boundaries; it reads what it sees; therefore, any file that is written by FOROTS will conform to the foregoing format requirements.

E.4.2 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data that may be referred to by either a READ or WRITE statement in the program being executed. A logical record is preceded by a control word and may have one or more control words embedded within it. In FORTRAN binary data files, there is no relationship between logical records and physical device block sizes. There is no implied maximum length for logical records.

E.4.2.1 Format of Binary Files – A FOROTS binary file may contain three forms of Logical Segment Control Words (LSCW). These LSCWs give FOROTS the ability to distinguish ASCII files from binary files.

	LSCW	
START	001+	the number of words in the segment (exclusive of any "END" LSCWs)
CONTINUE	002	indicates that the segment of a 128-word block boundary continues
END	003+	number of words in the preceding segment including LSCWs.

If the access specified for a file (through the OPEN statement ACCESS = parameter) is 'SEQIN', 'SEQOUT', or 'SEQINOUT', all three LSCWs may appear in a record. If the access specified is 'RANDIN', or 'RANDOM', all records are of the same length, and there are no CONTINUE LSCWs.

The following examples illustrate the LSCW. The random access binary file contains only 001 and 003 LSCW's.

```
C       LOOK AT A BINARY FILE AND SEE THE LOGICAL SEGMENT
C       CONTROL WORDS.

        OPEN(UNIT=1,ACCESS='RANDOM',MODE='BINARY',
     1       RECORD=100)

        I=5
        WRITE(1'1) (I, J=1,100)

        J=7
        WRITE(1'2) (J,K=1,100)
        END
```

```
000000  001000  000145  ← Number of words
000001  000000  000005    in record counting
000002  000000  000005    END LSCW or the
000003  000000  000005    number of words
000004  000000  000005    following this
000005  000000  000005    word to the
000006  000000  000005    END LSCW.
000007  000000  000005
000010  000000  000005
000011  000000  000005
000012  000000  000005
000013  000000  000005
000014  000000  000005
000015  000000  000005
000016  000000  000005
000017  000000  000005
000020  000000  000005
000021  000000  000005
000022  000000  000005
000023  000000  000005
000024  000000  000005
000025  000000  000005
000026  000000  000005
000027  000000  000005
000030  000000  000005
000031  000000  000005
000032  000000  000005
000033  000000  000005
000034  000000  000005
000035  000000  000005
000036  000000  000005
000037  000000  000005
000040  000000  000005
000041  000000  000005
000042  000000  000005
000043  000000  000005
000044  000000  000005
000045  000000  000005
000046  000000  000005
000047  000000  000005
000050  000000  000005
000051  000000  000005
000052  000000  000005
000053  000000  000005
000054  000000  000005
000055  000000  000005
000056  000000  000005
000057  000000  000005
000060  000000  000005
000061  000000  000005
000062  000000  000005
000063  000000  000005
000064  000000  000005
000065  000000  000005
000066  000000  000005
000067  000000  000005
000070  000000  000005
000071  000000  000005
000072  000000  000005
000073  000000  000005
000074  000000  000005
000075  000000  000005
000076  000000  000005
000077  000000  000005
000100  000000  000005
000101  000000  000005
000102  000000  000005
000103  000000  000005
000104  000000  000005
000105  000000  000005
000106  000000  000005
000107  000000  000005
000110  000000  000005
000111  000000  000005
000112  000000  000005
000113  000000  000005
000114  000000  000005
000115  000000  000005
000116  000000  000005
000117  000000  000005
000120  000000  000005
000121  000000  000005
000122  000000  000005
000123  000000  000005
000124  000000  000005
000125  000000  000005
000126  000000  000005
000127  000000  000005
000130  000000  000005
000131  000000  000005
000132  000000  000005
000133  000000  000005
000134  000000  000005
000135  000000  000005
000136  000000  000005
000137  000000  000005
000140  000000  000005
000141  000000  000005
000142  000000  000005
000143  000000  000005
000144  000000  000005
000145  003000  000146  ← END LSCW
000146  001000  000145    Containing the
000147  000000  000007    number of words
000150  000000  000007    in the record
                          including LSCW's.
```

```
000151  000000  000007
000152  000000  000007
000153  000000  000007
000154  000000  000007
000155  000000  000007
000156  000000  000007
000157  000000  000007
000160  000000  000007
000161  000000  000007
000162  000000  000007
000163  000000  000007
000164  000000  000007
000165  000000  000007
000166  000000  000007
000167  000000  000007
000170  000000  000007
000171  000000  000007
000172  000000  000007
000173  000000  000007
000174  000000  000007
000175  000000  000007
000176  000000  000007
000177  000000  000007
000200  000000  000007
000201  000000  000007
000202  000000  000007
000203  000000  000007
000204  000000  000007
000205  000000  000007
000206  000000  000007
000207  000000  000007
000210  000000  000007
000211  000000  000007
000212  000000  000007
000213  000000  000007
000214  000000  000007
000215  000000  000007
000216  000000  000007
000217  000000  000007
000220  000000  000007
000221  000000  000007
000222  000000  000007
000223  000000  000007
000224  000000  000007
000225  000000  000007
000226  000000  000007
000227  000000  000007
000230  000000  000007
000231  000000  000007
000232  000000  000007
000233  000000  000007
000234  000000  000007
000235  000000  000007
000236  000000  000007
000237  000000  000007
000240  000000  000007
000241  000000  000007
000242  000000  000007
000243  000000  000007
000244  000000  000007
000245  000000  000007
000246  000000  000007
000247  000000  000007
000250  000000  000007
000251  000000  000007
000252  000000  000007
000253  000000  000007
000254  000000  000007
000255  000000  000007
000256  000000  000007
000257  000000  000007
000260  000000  000007
000261  000000  000007
000262  000000  000007
000263  000000  000007
000264  000000  000007
000265  000000  000007
000266  000000  000007
000267  000000  000007
000270  000000  000007
000271  000000  000007
000272  000000  000007
000273  000000  000007
000274  000000  000007
000275  000000  000007
000276  000000  000007
000277  000000  000007
000300  000000  000007
000301  000000  000007
000302  000000  000007
000303  000000  000007
000304  000000  000007
000305  000000  000007
000306  000000  000007
000307  000000  000007
000310  000000  000007
000311  000000  000007
000312  000000  000007
000313  003000  000146
```

In the sequential access binary file the second record crosses the 128-word boundary and contains a 002 (CONTINUE) LSCW.

```
C       LOOK AT A BINARY FILE AND SEE THE LOGICAL SEGMENT
C       CONTROL WORDS.

        OPEN(UNIT=1,MODE='BINARY')

        I=5
        WRITE(1) (I, J=1,100)

        J=7
        WRITE(1) (J,K=1,100)
        END
```

```
000000  001000 000145
000001  000000 000005
000002  000000 000005
000003  000000 000005
000004  000000 000005
000005  000000 000005
000006  000000 000005
000007  000000 000005
000010  000000 000005
000011  000000 000005
000012  000000 000005
000013  000000 000005
000014  000000 000005
000015  000000 000005
000016  000000 000005
000017  000000 000005
000020  000000 000005
000021  000000 000005
000022  000000 000005
000023  000000 000005
000024  000000 000005
000025  000000 000005
000026  000000 000005
000027  000000 000005
000030  000000 000005
000031  000000 000005
000032  000000 000005
000033  000000 000005
000034  000000 000005
000035  000000 000005
000036  000000 000005
000037  000000 000005
000040  000000 000005
000041  000000 000005
000042  000000 000005
000043  000000 000005
000044  000000 000005
000045  000000 000005
000046  000000 000005
000047  000000 000005
000050  000000 000005
000051  000000 000005
000052  000000 000005
000053  000000 000005
000054  000000 000005
000055  000000 000005
000056  000000 000005
000057  000000 000005
000060  000000 000005
000061  000000 000005
000062  000000 000005
000063  000000 000005
000064  000000 000005
000065  000000 000005
000066  000000 000005
000067  000000 000005
000070  000000 000005
000071  000000 000005
000072  000000 000005
000073  000000 000005
000074  000000 000005
000075  000000 000005
000076  000000 000005
000077  000000 000005
000100  000000 000005
000101  000000 000005
000102  000000 000005
000103  000000 000005
000104  000000 000005
000105  000000 000005
```

```
000106  000000 000005
000107  000000 300005
000110  000000 000005
000111  000000 000005
000112  000000 000005
000113  000000 000005
000114  000000 000005
000115  000000 000005
000116  000000 000005
000117  000000 000005
000120  000000 000005
000121  000000 000005
000122  000000 000005
000123  000000 000005
000124  000000 000005
000125  000000 000005
000126  000000 000005
000127  000000 000005
000130  000000 000005
000131  000000 000005
000132  000000 000005
000133  000000 000005
000134  000000 000005
000135  000000 000005
000136  000000 000005
000137  000000 000005
000140  000000 000005
000141  000000 000005
000142  000000 000005
000143  000000 000005
000144  000000 000005
000145  003000 000146
000146  001000 000032  ← Number of words to next LSCW.
000147  000000 000007
000150  000000 000007
000151  000000 000007
000152  000000 000007
000153  000000 000007
000154  000000 000007
000155  000000 000007
000156  000000 000007
000157  000000 000007
000160  000000 000007
000161  000000 000007
000162  000000 000007
000163  000000 000007
000164  000000 000007
000165  000000 000007
000166  000000 000007
000167  000000 000007
000170  000000 000007
000171  000000 000007
000172  000000 000007
000173  000000 000007
000174  000000 000007
000175  000000 000007
000176  000000 000007
000177  000000 000007
000200  002000 000114  ← Continue LSCW.
000201  000000 000007
000202  000000 000007
000203  000000 000007
000204  000000 000007
000205  000000 000007
000206  000000 000007
000207  000000 000007
000210  000000 000007
000211  000000 000007
000212  000000 000007
000213  000000 000007
000214  000000 000007
000215  000000 000007
000216  000000 000007
000217  000000 000007
000220  000000 000007
000221  000000 000007
000222  000000 000007
000223  000000 000007
000224  000000 000007
000225  000000 000007
000226  000000 000007
000227  000000 000007
000230  000000 000007
000231  000000 000007
000232  000000 000007
000233  000000 000007
000234  000000 000007
000235  000000 000007
000236  000000 000007
000237  000000 000007
000240  000000 000007
000241  000000 000007
000242  000000 000007
000243  000000 000007
000244  000000 000007
000245  000000 000007
000246  000000 000007
000247  000000 000007
000250  000000 000007
000251  000000 000007
000252  000000 000007
000253  000000 000007
000254  000000 000007
000255  000000 000007
000256  000000 000007
000257  000000 000007
```

```
000260  000000 000007
000261  000000 000007
000262  000000 000007
000263  000000 000007
000264  000000 000007
000265  000000 000007
000266  000000 000007
000267  000000 000007
000270  000000 000007
000271  000000 000007
000272  000000 000007
000273  000000 000007
000274  000000 000007
000275  000000 000007
000276  000000 000007
```

```
000277  000000 000007
000300  000000 000007
000301  000000 000007
000302  000000 000007
000303  000000 000007
000304  000000 000007
000305  000000 000007
000306  000000 000007
000307  000000 000007
000310  000000 000007
000311  000000 000007
000312  000000 000007
000313  000000 000007
000314  003000 000147
```

Image mode files contain no LSCW's. This file cannot be backspaced.

```
C       LOOK AT AN IMAGE MODE FILE AND SEE NO LOGICAL SEGMENT
C       CONTROL WORDS.

        OPEN(UNIT=1,MODE='IMAGE')

        I=5
        WRITE(1) (I, J=1,100)

        J=7
        WRITE(1) (J,K=1,100)
        END
```

```
000000  000000 000005
000001  000000 000005
000002  000000 000005
000003  000000 000005
000004  000000 000005
000005  000000 000005
000006  000000 000005
000007  000000 000005
000010  000000 000005
000011  000000 000005
000012  000000 000005
000013  000000 000005
000014  000000 000005
000015  000000 000005
000016  000000 000005
000017  000000 000005
000020  000000 000005
000021  000000 000005
000022  000000 000005
000023  000000 000005
```

```
000024  000000 000005
000025  000000 000005
000026  000000 000005
000027  000000 000005
000030  000000 000005
000031  000000 000005
000032  000000 000005
000033  000000 000005
000034  000000 000005
000035  000000 000005
000036  000000 000005
000037  000000 000005
000040  000000 000005
000041  000000 000005
000042  000000 000005
000043  000000 000005
000044  000000 000005
000045  000000 000005
000046  000000 000005
000047  000000 000005
```

```
000050  000000  000005
000051  000000  000005
000052  000000  000005
000053  000000  000005
000054  000000  000005
000055  000000  000005
000056  000000  000005
000057  000000  000005
000060  000000  000005
000061  000000  000005
000062  000000  000005
000063  000000  000005
000064  000000  000005
000065  000000  000005
000066  000000  000005
000067  000000  000005
000070  000000  000005
000071  000000  000005
000072  000000  000005
000073  000000  000005
000074  000000  000005
000075  000000  000005
000076  000000  000005
000077  000000  000005
000100  000000  000005
000101  000000  000005
000102  000000  000005
000103  000000  000005
000104  000000  000005
000105  000000  000005
000106  000000  000005
000107  000000  000005
000110  000000  000005
000111  000000  000005
000112  000000  000005
000113  000000  000005
000114  000000  000005
000115  000000  000005
000116  000000  000005
000117  000000  000005
000120  000000  000005
000121  000000  000005
000122  000000  000005
000123  000000  000005
000124  000000  000005
000125  000000  000005
000126  000000  000005
000127  000000  000005
000130  000000  000005
000131  000000  000005
000132  000000  000005
000133  000000  000005
000134  000000  000005
000135  000000  000005
000136  000000  000005
000137  000000  000005
000140  000000  000005
000141  000000  000005
000142  000000  000005
000143  000000  000005
000144  000000  000007
000145  000000  000007
000146  000000  000007
000147  000000  000007
000150  000000  000007
000151  000000  000007
000152  000000  000007
000153  000000  000007
000154  000000  000007
000155  000000  000007
000156  000000  000007
000157  000000  000007
000160  000000  000007
000161  000000  000007
000162  000000  000007
000163  000000  000007
000164  000000  000007
000165  000000  000007
000166  000000  000007
000167  000000  000007
000170  000000  000007
000171  000000  000007
000172  000000  000007
000173  000000  000007
000174  000000  000007
000175  000000  000007
000176  000000  000007
000177  000000  000007
000200  000000  000007
000201  000000  000007
000202  000000  000007
000203  000000  000007
000204  000000  000007
000205  000000  000007
000206  000000  000007
000207  000000  000007
000210  000000  000007
000211  000000  000007
000212  000000  000007
000213  000000  000007
000214  000000  000007
000215  000000  000007
000216  000000  000007
000217  000000  000007
000220  000000  000007
000221  000000  000007
```

```
000222  000000 000007
000223  000000 000007
000224  000000 000007
000225  000000 000007
000226  000000 000007
000227  000000 000007
000230  000000 000007
000231  000000 000007
000232  000000 000007
000233  000000 000007
000234  000000 000007
000235  000000 000007
000236  000000 000007
000237  000000 000007
000240  000000 000007
000241  000000 000007
000242  000000 000007
000243  000000 000007
000244  000000 000007
000245  000000 000007
000246  000000 000007
000247  000000 000007
000250  000000 000007
000251  000000 000007
000252  000000 000007
000253  000000 000007
000254  000000 000007
000255  000000 000007
000256  000000 000007
000257  000000 000007
000260  000000 000007
000261  000000 000007
000262  000000 000007
000263  000000 000007
000264  000000 000007
000265  000000 000007
000266  000000 000007
000267  000000 000007
000270  000000 000007
000271  000000 000007
000272  000000 000007
000273  000000 000007
000274  000000 000007
000275  000000 000007
000276  000000 000007
000277  000000 000007
000300  000000 000007
000301  000000 000007
000302  000000 000007
000303  000000 000007
000304  000000 000007
000305  000000 000007
000306  000000 000007
000307  000000 000007
```

E.4.3 Mixed Mode Data Files

FOROTS permits files containing both ASCII and binary data records to be read. Mixed files may be accessed in either sequential or random access mode. Logical ASCII and binary records have the same format as described in the preceding paragraphs. In random access mode, the record size must be large enough to contain the largest record either ASCII or binary.

E.4.4 Image Files

The image data transfer mode is a buffered mode in which data is transferred in a blocked format consisting of a word count located in the right half of the first data word of the buffer followed by the number of 36-bit data words. The devices which permit image data transfers and the form in which the data is read or written are:

Device	Data Forms
Card Reader	All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte contains column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. Cards are not split between two buffers.
Disk	Data is written on the disk exactly as it appears in the buffer. Data consists of 36-bit words.

Device	Data Forms
Magnetic Tape	Data appears on magnetic tape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation.

E.5 USING FOROTS

FOROTS has been designed to lend itself for use as an I/O system for programs written in languages other than FORTRAN. Currently, MACRO programmers may employ FOROTS as a general I/O system by writing simple MACRO calls which simulate the calls made to FOROTS by a FORTRAN compiler. The calls made to FOROTS are to routines that implement FORTRAN I/O statements such as READ, WRITE, OPEN, CLOSE, RELEASE, etc.

FOROTS provides automatic memory allocation, data conversion, I/O buffering, and device interface operations to the MACRO user.

E.5.1 FOROTS Entry Points

FOROTS provide the following entry points for calls from either a FORTRAN compiler or a non-FORTRAN program:

Entry Point	Function
ALCHN.	Allocate software channels
ALCOR.	Allocate dynamic memory blocks
CLOSE.	Close a file
DEC.	DECODE routine
DECHN.	De-allocate software channels
DECOR.	De-allocate dynamic memory blocks
ENC.	ENCODE routine
EXIT.	Terminate program execution
FIN.	Input/Output list termination routine
FIND.	Position to the next record (RANDOM ACCESS)
FORER.	Error processor
IN.	Formatted input routine
IOLST.	Input/Output list routine
MTOP.	File utility processing routine
NLI.	NAMELIST input routine
NLO.	NAMELIST output routine
OPEN.	Open a file
OUT.	Formatted output routine
RELEA.	Release a device (CLOSE implied)
RESET.	Job initialization entry
RTB.	Binary input routine
TRACE.	Trace subroutine calls
WTB.	Binary output routine

E.5.2 Calling Sequences

All calls made to FOROTS must be made using the following general form:

```
MOVEI   16,ARGBLK
PUSHJ   17,Entry Point
        (control is returned here)
```

where:

a. ARGBLK is the address of a specifically formatted argument block which contains information needed by FOROTS to accomplish the desired operation.

b. Entry Point is an entry point identifier (see list given in Paragraph E.6.1) which specifies the entry point of the desired FOROTS routine.

With three exceptions, all returns from FOROTS will be made to the program instruction immediately following the call (PUSHJ 17, entry point instruction). The exceptions are:

a. An error return to a specified statement number (i.e., READ or WRITE statement ERR= option),

b. An end-of-file return to a statement number (i.e., READ or WRITE statement END= option),

c. A fatal error which returns to the monitor or to a debug package.

Paragraphs E.5.3.1 through E.5.3.11 give the MACRO calls and required argument block formats needed to initialize FOROTS and FOROTS I/O operations.

Argument blocks conform to the subprogram calling convention described in Appendix D. However, there is one exception in dealing with the first word of an I/O initialization call (i.e., WTB., ENC., RTW., etc.) for a FORTRAN logical unit number. If the type field of the first word of an I/O initialization call for the FORTRAN logical unit number is 0 (zero), the argument is an immediate mode (18 bit) constant wherever possible. If the type field is integer, the argument is indirect (see Appendix D, Table D-2, Type 2).

This exception should not cause any upward compatibility problems since all previously working programs will still function. An added feature with this convention is that it permits the following construct to be correctly implemented:

```
          N = -4                    !SET FOR TERMINALS
          READ (N,100) I,J
100       FORMAT(2I5)
```

E.5.3 MACRO Calls for FOROTS Functions

The forms of the MACRO calls to FOROTS that are made by the FORTRAN compiler are described in the following paragraphs. The calls described are identified according to the language statement which they implement. The following terms and abbreviations may be used in the description of the argument block (ARGBLK) of each call:

→ = pointer to the second word in the argument block (This is the address pointed to by the argument ARGBLK in the calling sequence),

n = count of ASCII characters,

f = FORMAT statement address,

v = the name of an array containing ASCII characters,

list = an Input/Output list,

c = the statement to which control is transferred on an "END OF FILE" condition,

d = the statement to which control is transferred on an "ERROR" condition,

name = a NAMELIST name,

R = a variable specifying the logical record number for random access mode,

* = list directed I/O; the FORMAT statement is not used,

type = type specification of a variable or constant,

where ARGBLK is

0_____8	9___12	13	14_____17	18_____35
-6				0
Reserved	2	I	X	n
	7	I	X	c
	7	I	X	d
	0	I	X	f
	2	I	X	Format Size (in words)
Reserved	0	I	X	v

E.5.3.1 **Formatted/Unformatted Transfer Statements, Sequential Access Calling Sequences** – The READ and WRITE statements for formatted sequential data transfer operations and their calling sequences are:

```
READ (u,f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```
WRITE (u,f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-5				0
Reserved	2	I	X	u
	7	I	X	c
	7	I	X	d
	0	I	X	f
Reserved	2	I	X	Format Size (in words)

The READ and WRITE statements for unformatted sequential data transfer operations and their calling sequences are:

```
READ (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.
```

and

```
WRITE (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

where ARGBLK is

0----8	9-----12	13	14---- 17	18----35
-3				0
Reserved	2	I	X	u
↓	7	I	X	c
Reserved	7	I	X	d

E.5.3.2 NAMELIST Data Transfer Statements, Sequential Access Calling Sequences – The READ and WRITE statements for namelist-directed sequential data transfer operations and their calling sequences are:

```
READ (u, name)
READ (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLI.
```

and

```
WRITE (u, name)
WRITE (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLO.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-4				0
Reserved	2	I	X	u
	7	I	X	c
	7	I	X	d
Reserved	0	I	X	Namelist table addr.

The NAMELIST table is generated from the FORTRAN NAMELIST. The first word of the table is the NAMELIST name; following that are a number of 2-word entries for scalar variables, and a number of (N+3)=word entries for array variables, where N is the dimensionality of the array.

The names specified in the NAMELIST statement are stored, in SIXBIT form, first in the table. Each name is followed by a list of arguments associated with the name; this argument list may be of any length and is terminated by a zero entry. The name argument list may be in either a scalar or an array form (refer to the following diagrams).

E.5.3.3 Array Offsets and Factoring – Address calculations used to reference a given array element involve factors and offsets. For example:

Array A is dimensioned

DIMENSION A (L1/U1,L2/U2,L3/U3,. . .Ln/Un)

The size of each dimension is represented by

S1 = U1-L1+1
S2 = U2-L2+1
etc.

In order to calculate the address of an element referenced by

A (Il,I2,I3,. . .In)

the following formula is used:

A+(I1-L1)+(I2-L2)*S1+(I3-L3)*S2*S1+. . .+(In-Ln)*S[n-1] *. . .*S2*S1

The terms are factored out depending on the dimensions of the array and not on the element referenced to arrive at the formula

A+(-L1-L2*S1-L3*S2*S1. . .)+I1+I2*S1+I3*S2*S1. . .

The parenthesized part of this formula is the offset for a single precision array and it is referred to as the Array Offset.

For each dimension of a given array, there is a corresponding factor by which a subscript in that position will be multiplied. From the last expression, one can determine the factor for dimension n to be

S[n-1]*S[n-2]*...*S2*S1

For double precision and complex arrays, the expression becomes

A+2*(I1-L1)+2*(I2-L2)*S1+2*(I3-L3)*S2+S1+...

Therefore, the array offset for a double precision array is

2*(-L1-L2*S1-L3*S2*S1...)

and the factor for the nth dimension is

2*S[n-1]*S[n-2]*...*S2*S1

The factor for the first dimension of a double precision array is always 2. The factor for the first dimension of a single precision array is always 1.

SCALAR ENTRY in a NAMELIST Table

0 • • • 8	9 • • • 11	12 • • • 14	15 • • • 17	18 • • • 35
SIXBIT/SCALAR NAME/				
0	0	I	X	Scalar addr

ARRAY ENTRY in a NAMELIST Table

0 • • • 8	9 • • • 11	12 • • • 14	15 • • • 17	18 • • • 35
SIXBIT/ARRAY NAME/				
#DIMS	type	I	X	
ARRAY SIZE				OFFSET
		I	X	Factor 1
		I	X	Factor 2
		I	X	Factor 3
		I	X	Factor n

E.5.3.4 Formatted/Unformatted Data Transfer Statements, Random Access Calling Sequences – The READ and WRITE statements for random access data transfer operations and their calling sequences are:

READ (u#R, f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.

and

```
WRITE (u#R, f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-6				0
Reserved	2	I	X	u
	7	I	X	c
	7	I	X	d
	0			0
	0			0
Reserved	2	I	X	address Record Number

E.5.3.5 Calling Sequences for Statements Which Use Default Devices – The FORTRAN statements that require the use of a reserved system default device and their calling sequences are:

	Default Device	
ACCEPT f, list	UNIT= -4	(TTY)
READ f, list	UNIT= -5	(CDR)
REREAD f, list	UNIT= -6	(REREAD)

```
MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-5				0
Reserved	2	I	X	u
	7	I	X	c
	7	I	X	d
	0	I	X	f
Reserved	2	I	X	Format Size (in words)

Default Device

PRINT f, list	UNIT= -3	(LPT)
TYPE f, list	UNIT= -1	(TTY)

```
MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-3				0
Reserved	2	I	X	u
↓	7	I	X	c
Reserved	7	I	X	d

E.5.3.6 Calling Sequences for Statements Which Position Magnetic Tape Units – The FORTRAN statements that may be used to control the positioning of a magnetic tape device and their calling sequences are:

Function (FORTRAN Statement)	FOROTS Code
SKIPFILE (u)	7
BACKFILE (u)	3
BACKSPACE (u)	2
ENDFILE (u)	4
REWIND (u)	0
SKIPRECORD (u)	5
UNLOAD (u)	1

CALL:

```
MOVEI 16, ARGBLK
PUSHJ 17, MTOP.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-3				0
Reserved	FOROTS code	I	X	u
↓		I	X	c
Reserved		I	X	d

E.5.3.7 **List Directed Input/Output Statements** – Any form of a formatted input/output statement may be written as a list-directed statement by replacing the referenced FORMAT statement number with an asterisk (*). The list-directed forms of the READ and WRITE statements and their calling sequences are:

```
READ (u, *, END=c, ERR=d) list
READ (u#R, *, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```
WRITE (u, *, END=c, ERR=d) list
WRITE (u#R, *, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-5				0
Reserved	2	I	X	u
	7	I	X	c
	7	I	X	d
	0	0	0	0
Reserved	0	0	0	0

E.5.3.8 **Input/Output Data Lists** – The compiler generates a calling sequence to the runtime system if an I/O list is defined for the READ or WRITE statement. The argument block associated with the calling sequence contains the addresses of the variables and arrays to be transferred to or from an I/O buffer. The general form of an I/O list calling sequence is:

```
MOVEI 16, ARGBLK
PUSHJ 17, IOLST.
```

Any number of elements may be included in the ARGBLK. The end of the argument block is specified by a zero entry or a call to the FIN. entry.

Mnemonic Name	FOROTS Value
DATA	1
SLIST	2
ELIST	3
FIN	4

The elements of an I/O list are:

1. DATA

   The DATA element converts one single, double, or complex precision item from external to internal form for a READ statement and from internal to external form for a WRITE statement. Each DATA element has the following format.

0_____8	9_____12	13	14_____17	18_____35
DATA	type	I	X	SCALAR ADDR

2. SLIST

   The SLIST. argument converts an entire array from internal to external form or vice versa depending on the type of statement (i.e., READ or WRITE) involved. An SLIST. table has the following form:

0_____ 8	9_____ 12	13	14_____17	18_____35
SLIST.		I	X	#ELEMENTS
		I	X	INCREMENTS
0	type	I	X	BASE ADDR1.

For example, the sequence:

```
DIMENSION A(100, B(100)
READ(–,–) A
        or
READ (–,–) (A(I),I=1,100)          ! only when the /OPT switch is used
```

develops an SLIST argument of the form:

0_____8	9_____12	13	14_____17	18_____35
0				
2	0	0	0	100
0	0	0	0	1
0	type	I	X	A

The increment may be zero. This could be produced by the sequence

```
DIMENSION A(100)
WRITE(–,–) (K,I=1,100)          ! only when the /OPT switch is used
```

The zero may not appear as an immediate constant in the argument block. The SLIST for the previous example would be

0_____8	9_____12	13	14_____17	18_____35
SLIST				100
		I		Pointer to a word containing a zero
				K

3. ELIST

The SLIST format permits only a single increment for a number of arrays to be specified while the ELIST permits different increments to be specified for different arrays.

The format of the ELIST is

0_____8	9_____12	13	14_____17	18_____35
ELIST				No. Elements to transfer increment 1
				Base ADDR 1 increment 2
				Base ADDR 2 increment N
				Base ADDR N

For example, the FORTRAN sequence

```
DIMENSION IC(6,100), IB(100)
WRITE(-,-) (IB(I),IC(1,I),I=1,100)
```

produces the ELIST

0_____8	9_____12	13	14_____17	18_____35
3				100
				1
	2			IB
				6
	2			IC

4. FIN

The end of an I/O list is indicated by a call to the FIN routine in the object time system. This call must be made after each I/O initialization call, including calls with a null I/O list. The FIN routine may be entered by an explicit call or by an argument in the I/O list argument block. If both calls are used, the explicit call has no meaning. The FIN element has the following format:

EXPLICIT CALL:

```
MOVEI 16,ZERBLK
PUSHJ 17,FIN.
```

where ZERBLK is

0 -- 35
0
0

E.5.3.9 **OPEN and CLOSE Statements, Calling Sequences** – The form and calling sequences for the OPEN and CLOSE FORTRAN statements are:

OPEN STATEMENT CALL

```
MOVEI 16, ARGBLK
PUSHJ 17, OPEN.
```

CLOSE STATEMENT CALL

```
MOVEI 16, ARGBLK
PUSHJ 17, CLOSE.
```

where ARGBLK is

0 ------ 8	9 ------ 12	13	14 ------ 17	18 ------ 35
Negative of the # of words in block not including this one.				0
→ G	2	I	X	u
G	7	I	X	c
G	7	I	X	d
G	type	I	X	H
G	type	I	X	H
G	type	I	X	H
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
G	type	I	X	H

The G field (bits 0–8) contains a 2-digit numeric which defines the argument name; the H field (bits 18–35) contains an address which points to the value of the argument.

The numeric codes which may appear in the G field and the argument which each identifies are

G Field	Open Argument	G Field	Open Argument
01	DIALOG	10	DIRECTORY
02	ACCESS	11	UNIT
03	DEVICE	12	MODE
04	BUFFER COUNT	13	FILE SIZE
05	BLOCK SIZE	14	RECORD SIZE
06	FILENAME	15	DISPOSE
07	PROTECTION	23	PARITY
		24	DENSITY

E.5.3.10 Software Channel Allocation and De-allocation Routines – Software channels may be allocated by MACRO programs via calls to the ALCHN. routine and de-allocated by calls to the DECHN. routine. Values are returned in AC 0.

The ALCHN. entry is used to allocate a particular channel or the next available channel. If bits 18–35 of the ARGBLK are zero, the next available channel will be assigned; if non-zero, they must contain the requested channel number (1–15 octal). If the channel requested is not available or all channels are in use, ALCHN. returns with -1 in AC0. Normal returns contain the assigned channel number in AC0.

The calling sequence of an ALCHN. routine is:

```
MOVEI 16, ARGBLK
PUSHJ 17, ALCHN.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-1				0
Reserved	type	I	X	Pointer to a word containing the channel# or zero

The DECHN. entry is used to de-allocate a previously assigned channel. The channel to be released is passed to DECHN. in the argument block variable. If the channel to be de-allocated was not assigned by ALCHN. and thus cannot be de-assigned, AC 0 is set to -1 on return.

The calling sequence for a DECHN. routine is:

```
MOVEI 16, ARGBLK
PUSHJ 17, DECHN.
```

where ARGBLK is

0_____8	9_____12	13	14_____17	18_____35
-1				0
Reserved	type	I	X	Pointer to a word containing the channel# to be released

E.6 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

FORTRAN logical and physical device assignments are made by the user at run time or standard system assignments are made according to a FOROTS Device Table (i.e., DEVTB.). The standard assignments contained by the Device Table are shown in Table E-2.

Table E-2
FORTRAN Device Table

Device/Function	FORTRAN Logical Unit Number	Use
REREAD	-6	REREAD statement
CDR	-5	READ statement
TTY	-4	ACCEPT statement
LPT	-3	PRINT statement
–	not valid	
TTY	-1	TYPE statement
0	00	ILLEGAL
DSK	01	DISK
CDR	02	Card Reader
LPT	03	Line Printer
CTY	04	Console Teletype
TTY	05	User's Teletype
	not valid	
MTA0	16	Magnetic Tape
MTA1	17	↓
MTA2	18	
FORTR	19	Assignable Device
DSK	20	DISK
DSK	21	↓
DSK	22	
DSK	23	
DSK	24	
DEV1	25	Assignable Devices
DEV2	26	↓
DEV3	27	
DEV4	28	
DEV5	29	

APPENDIX F
FORDDT

FORDDT is an interactive program used to debug FORTRAN programs and control their execution. By using the symbols created by the FORTRAN compiler, FORDDT allows the user to examine and modify the data and FORMAT statements in his program, set breakpoints at any executable statement or routine, trace his program statement by statement, and make use of many other debugging techniques described in this appendix.

Table F-1 provides a brief glance at all the commands available to the user of FORDDT.

Table F-1
Table of Commands

Command	Purpose
Data Access Commands	
ACCEPT	Modifies data locations.
TYPE	Displays data locations.
Declarative Commands	
GROUP	Defines indirect lists for TYPE statements.
MODE	Specifies format of typeout.
OPEN	Accesses program unit symbol table.
PAUSE	Places pause requests.
REMOVE	Removes pause requests.
DIMENSION	Defines dimensions of arrays for FORDDT references. (Unnecessary if /DEBUG:DIMENSIONS was used.)
DOUBLE	Defines dimensions of double precision arrays for FORDDT references. (Unnecessary if /DEBUG:DIMENSIONS was used.)

Table F-1 (Cont)
Table of Commands

Command	Purpose
Control Commands	
START	Begins execution of FORTRAN program.
CONTINUE	Continues execution after a pause.
GOTO	Transfers control to some program statement within the open program unit.
NEXT	Traces execution of the program.
STOP	Terminates program and returns to monitor mode.
DDT	Enters DDT (if DDT is loaded).
Other Commands	
LOCATE	Lists program unit names in which a given symbol is defined.
STRACE	Displays routine backtrace of current program status.
WHAT	Displays current DIMENSION, GROUP, and PAUSE information.

F.1 INPUT FORMAT

FORDDT commands are made up of alphabetic FORTRAN-like identifiers and need consist of only those characters that are required to make the command unique. If the user wishes to specify parameters, a space or tab is required following the command name. FORDDT expects a parameter if a delimiter is found.

F.1.1 Variables and Arrays

FORDDT allows the user to access and modify the data locations in his program by using standard DECsystem-20 FORTRAN symbolic names. Variables are specified simply by name. Array elements are specified in the following format:

name $(S_1,\ldots,S_n)$

where

name = a FORTRAN variable or array name
$(S_1,\ldots,S_n)$ = the subscripts of the particular array.

An entire array may be referenced simply by its unsubscripted name; a range of array elements may be specified by inputting the first and last array elements of the desired range, separated by a dash (–).

Examples

ALPHA
ALPHA(7)
ALPHA(PI)
ALPHA(2)–ALPHA(5)

F.1.2 Numeric Conventions

FORDDT accepts optionally signed numeric data in the standard FORTRAN input formats:

1. INTEGER – A string of decimal digits.

2. FLOATING POINT – A string of decimal digits optionally including a decimal point. Standard engineering and double precision exponent formats are also accepted.

3. OCTAL – A string of octal digits optionally preceded by a double quote (").

4. COMPLEX – An ordered pair of integer or real constants separated by a comma and enclosed in parentheses.

F.1.3 Statement Labels and Source Line Numbers

FORTRAN statement labels are input and output by straightforward numeric reference (i.e., 1234). However, source line numbers must be input to FORDDT with a number sign (#) preceding them. This mandatory sign distinguishes statement labels from source line numbers.

F.2 NEW USER TUTORIAL

The new FORDDT user can rely on the commands described below as a basis for debugging FORTRAN programs. The new user will find these commands easy to understand and apply.

F.2.1 Basic Commands

The easiest method of loading and starting FORDDT is

```
@DEBUG filename.type (DEBUG), SYS.FORDDT.REL
```

FORDDT will respond with

```
ENTERING FORDDT
>>
```

Just as an asterisk(*) signifies FORTRAN's readiness, the two angle brackets signify that FORDDT is awaiting one of the following commands:

OPEN — Makes available to FORDDT the symbol names in a particular program unit of the FORTRAN program. When a program unit symbol table is opened, the previously open program unit is automatically closed. When FORDDT is entered, the MAIN program is automatically opened. The command format is

```
OPEN name
```

This will open the particular program unit named and allow all variables within that subprogram to be accessible to FORDDT.

```
OPEN
```

with no arguments will reopen the symbol table of the main program unit.

START — Starts the user program at the main program entry point. The command format is

```
START
```

STOP

Terminates program execution, causes all files to be closed, and exits to the monitor. The command format is

STOP

MODE

Defines the display format for succeeding FORDDT TYPE commands. Only the first character of the mode need be typed to identify it to FORDDT. The modes are

Mode	Meaning
F	FLOATING POINT
D	DOUBLE PRECISION
C	COMPLEX
I	INTEGER
O	OCTAL
A	ASCII (left-justified)
R	RASCII (right-justified)

Unless the MODE command is given, the default typeout mode is the floating point format.

The command format is

MODE list

where list contains one or more of the mode identifiers separated by commas. The current setting can be changed by issuing another MODE command. If more than one mode is given, the values are typed out in the order: F,D,C,I,O,A,R

MODE

with no arguments will reset FORDDT to the original setting of floating point format.

TYPE

Allows the user to display the contents of one or more data locations. They are displayed on the user terminal formatted according to the last MODE specification. The command format is

TYPE list

where list may contain one or more arrays, variables, array elements, or array element ranges separated by commas. For example

TYPE I, ALPHA, BETA(2), J(3)–J(5)

Each item will be displayed in each of the currently active typeout modes as set by the last MODE command.

ACCEPT

Allows the user to change the contents of a FORTRAN variable, array, array element, or array element range. The command format is

ACCEPT name/mode value

where

name = the name of the variable, array, array element, or array element range to be modified. If the field contains an unsubscripted array name or an element range, it causes all of the elements to be set to the given value (see special case for ASCII in section F.6).

mode = the format of the data value to be entered. If given, it must be preceded by a slash (/) and immediately follow the name. (Note that /mode does not apply to FORMAT modification.)

value = the new value to be assigned. It must correspond in format to the given mode.

Data Modes

Only the first character of a data mode need be typed to identify it to FORDDT. If not specified, the default mode is REAL. The following input modes are available:

Mode	Meaning	Example
A	ASCII (left-justified)	/FOO/
C	COMPLEX	(1.25,-78.E+9)
D	DOUBLE PRECISION	123.4567890
F	REAL	123.45678
I	INTEGER	1234567890
O	OCTAL	76543210
R	RASCII (right-justified)	\BAR\
S	SYMBOLIC	PSI(2,4)

An example of the ACCEPT command format is

```
ACCEPT ALPHA 100.6
```

This changes the value of the variable ALPHA to 100.6 with the default input mode of REAL, since mode was not specified.

PAUSE

Allows the user to set a breakpoint at any label, line number, or subroutine entry in the user program. Up to 10 pauses may be set at one time. When one of these pauses is encountered, execution of the FORTRAN program is suspended and control is transferred to FORDDT. Also, when a pause is encountered, the symbol table of that subprogram is automatically opened. The command format is

```
PAUSE P
```

where P is a statement label number, line number, or routine entry point name; for example

```
PAUSE 100
```

will cause a breakpoint at statement label 100 of the currently open program unit.

Note that subprogram parameter values will be displayed when a pause is encountered at a subprogram entry point.

CONTINUE — Allows the program to resume execution after a FORDDT pause. After a CONTINUE is executed, the program either runs to completion, or it runs until another pause is encountered. If a value is included with this command, the program will run until the nth occurrence of the given pause or until a different pause is encountered. The command formats are

```
CONTINUE
  or
CONTINUE n
```

Example

```
CONTINUE 15
```

will continue execution until the fifteenth occurrence of the pause.

REMOVE — Used to remove those pauses from the program previously set up by the PAUSE command. The command format is

```
REMOVE P
```

where P is the number of the statement label where the pause was set, i.e.,

```
REMOVE 100
```

will remove the pause at statement label 100.

Note that REMOVE with no arguments will remove all pauses; therefore, no abbreviation of the command is allowed in this instance. This precaution prevents the accidental removal of all pauses.

WHAT — Displays on the user terminal the name of the currently open program unit and any currently active pause settings. The command format is

```
WHAT
```

F.3 FORDDT AND THE FORTRAN/DEBUG SWITCH

Most facilities of FORDDT are available without the FORTRAN/DEBUG features; however, if the/DEBUG switch is not used when compiling a FORTRAN program, the trace features (NEXT command) will not be available, and several of the other commands will be restricted.

Using the /DEBUG switch tells FORTRAN to compile extra information for FORDDT. (See Appendix C Using the Compiler for a complete description of each feature.) The additional features include

1. /DEBUG:DIMENSIONS which will generate dimension information to the REL file for all arrays dimensioned in the subprogram. The dimension information will automatically be available to FORDDT if the user wishes to reference an array in a TYPE or ACCEPT command. This feature eliminates the need to specify dimension information for FORDDT by using the DIMENSION command.

2. /DEBUG:LABELS which will generate labels for every executable source line in the form "line-number L". If these labels are generated, they may be used as arguments with the FORDDT commands PAUSE and GOTO.

   This switch will also generate labels at the last location allocated for a FORMAT statement so that FORDDT can detect the end of the statement. These labels have the form "format-label F". If they are generated, the user will be able to display and modify his FORMAT statements via the TYPE and ACCEPT commands.

   Note that the :LABELS switch is automatically activated with the :TRACE switch since labels are needed to accomplish the trace features.

3. /DEBUG:TRACE which will generate a reference to FORDDT before each executable statement. This switch is required in order for the trace command NEXT to function.

   Note that if more than one FORTRAN statement has been placed on a single input line, only the first statement will have a FORDDT reference and line-number label associated with it. This also applies to the :LABELS switch.

4. /DEBUG:INDEX which will force the compiler to store, in its respective data location as well as a register, the index variable of all DO loops at the beginning of each loop iteration. The user will then be able to examine DO loops by using FORDDT. If a user modifies a DO loop index using FORDDT, he will not affect the number of loop iterations because a separate loop count is used (see section D.1.5).

   Note that this switch has no direct affect on any of the commands in FORDDT.

F.4 LOADING AND STARTING FORDDT

1. The simplest form of loading and starting FORDDT is with the following command string:

   ```
   @DEBUG (FROM) filename.type/FORTRAN
   ```

   FORDDT responds with

   ```
   ENTERING FORDDT
   >>
   ```

   The angle brackets indicate that FORDDT is ready to receive a command, just like an asterisk (*) signifies FORTRAN's readiness.

   The DEBUG command to the monitor will also load DDT (standard system debugging program). DDT can be used or ignored but it does require an extra 4 pages of memory.

2. The user may wish to load his compiled program and FORDDT directly with the LINK program. (Loading with LINK was accomplished implicity in the previous command string.) The command sequence is as follows:

   ```
   @ LINK
   *filename.type/DEBUG:FORTRAN/60
   */SYMSEG/G
   ```

   If the total FORTRAN program consists of many subroutines and insufficient memory is available to complete loading with symbols, it is possible to load with symbols just those sections expected to give trouble. The remaining routines need not be loaded.

F.5 SCOPE OF NAME AND LABEL REFERENCES

Each program unit has its own symbol table. When the user initially enters FORDDT, he automatically opens the symbol table of the main program. All references to names or labels via FORDDT must be made with respect to the currently open symbol table. If the user has given the main program a name other than MAIN by using the PROGRAM statement (see Chapter 5, section 5.2), FORDDT will ask for the defined program name. After the user enters the program name, FORDDT will open the appropriate symbol table. At this point, symbol tables in programs other than the main program can be opened by using the OPEN command (see section F.6).

References to statement labels, line numbers, FORMAT statements, variables, and arrays must have labels that are defined in the currently open symbol table. However, FORDDT will accept variable and array references outside the currently open symbol table providing the name is unique with respect to all program units in the given load module.

F.6 FORDDT COMMANDS

This section gives a detailed description of all commands in FORDDT. The commands are given in alphabetical order.

ACCEPT

Allows the user to change the contents of a FORTRAN variable, array, array element, array element range, or FORMAT statement. The command format is

ACCEPT name/mode value

where

name = the variable array, array element, array element range, or FORMAT statement to be modified.

mode = the format of the data value to be entered. The mode keyword must be preceded by a slash (/) and immediately follow the name. Intervening blanks are not allowed. (Note that /mode does not apply to FORMAT modification.)

value = the new value to be assigned. The format of the input value must correspond to the specified mode.

DATA LOCATION MODIFICATION

Data Modes

The following data modes are accepted:

Mode	Meaning	Example
A	ASCII (left-justified)	/FOO/
C	COMPLEX	(1.25,-78.E+9)
D	DOUBLE PRECISION	123.4567890
F	REAL	123.45678
I	INTEGER	1234567890
O	OCTAL	76543210
R	RASCII (right-justified)	\BAR\
S	SYMBOLIC	PSI(2,4)

If not specified, the default mode is REAL.

Two Word Values

For the data modes ASCII, RASCII, OCTAL, and SYMBOLIC, FORDDT will accept a "/LONG" modifier on the mode switch. This modifier indicates that the variable and the value are to be interpreted as two words in length.

Example

```
ACCEPT VAR/RASCII/LONG '1234567890'
```

will assume that VAR is two words long and store the given 10 character literal into it.

Initialization of Arrays

If the name field of an ACCEPT contains an unsubscripted array name or a range of array elements, all elements of the array or the specified range will be set to the given value.

Example

```
ACCEPT ARRAY/F 1.0
          or
ACCEPT ARRAY(5)–ARRAY(10)/F 1.0
```

Note that this applies only to modes other than ASCII and RASCII.

Long Literals

When the value field of an ACCEPT contains an unsubscripted array name or range of array elements, and the specified data mode is ASCII or RASCII, the value field is expected to contain a long literal string. ACCEPT will store the string linearly into the array or array range. If the array is not filled, the remainder of the array or range will be set to zero. If the literal is too long the remaining characters will be ignored.

Example

```
ACCEPT ARRAY/RASCII 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

FORMAT STATEMENT MODIFICATION

When the name field of an ACCEPT contains a label, it expects this label to be a FORMAT statement label and that the value field contains a new FORMAT specification.

Example

```
ACCEPT 10 (1H0,F10.2,3(I2))
```

The new specification cannot be longer than the space originally allocated to the FORMAT by the compiler. The remainder of the area is cleared if the new specification is shorter.

Note that FOROTS performs some amount of encoding of FORMAT statements when it processes them for the first time. If any I/O statement referencing the given FORMAT has been executed, the FORTRAN program has to be restarted (re-initializing FOROTS).

CONTINUE

Allows the program to resume execution after a FORDDT pause. After a CONTINUE is executed, the program either runs to completion or until another pause is encountered. The command format is

CONTINUE n

where the n is optional and if omitted will be assumed to be one. If a value is provided, it may be a numeric constant or program variable but it will be treated as an integer. When the value n is specified, the program will continue execution until the nth occurrence of this pause. For example,

CONTINUE 20

will continue execution after the 20th occurrence of the pause.

DIMENSION

Sets the user defined dimensions of an array for FORDDT access purposes. These dimensions need not agree with those declared to the compiler in the source code. FORDDT will allow the user to redimension an array to have a larger scope than that of the source program. If this is done a warning is given. The command format is

DIMENSION S

For example:

DIMENSION ALPHA(7,5/6.10)

where S is the name of the array specified.

FORDDT will remember the dimensions of the array until it is redefined or removed.

The command

DIMENSION

will give a full list of all the user defined dimensions for all arrays.

DIMENSION ALPHA

will display the current information for the array ALPHA only.

DIMENSION ALPHA/REMOVE

will remove any user defined array information for the array ALPHA.

Arrays, Array Elements, and Ranges

Array elements are specified in the following format:

name $[d_1/d_2,\ldots](S_1,\ldots)$

where

name = the name of the array

[. . .] = optional, and contains dimension information. This form is equivalent in effect to the DIMENSION statement.

(. . .) = the subscripts of the specific element desired.

The entire array is referenced simply by its unsubscripted name. A range of array elements is specified by inputting the first and last array elements of the desired range separated by a dash (–) (A(5)–A(10)).

DOUBLE

Defines the dimensions of a double precision array. The result of this command is the same as for the DIMENSION command except that the array so dimensioned is understood by FORDDT to be an array with two word entries and, therefore, reserves twice the space. The command format is

DOUBLE arrayname

GOTO

Allows the user to continue his program from a point other than the one at which he last paused. The GOTO allows the user to continue at a statement label or code generating source line number provided that the /DEBUG:LABELS switch has been used or the contents of a symbol previously ASSIGNed during the program execution.

Note that the program must be STARTed before this command can be used and also note that a GOTO is not allowed after the CTRL/C CTRL/C REENTER sequence. (See F.6.)

The command format is

GOTO n

GROUP

Sets up a string of text for input to a TYPE command. The user can store TYPE statements as a list of variables identified by the numbers 1 through 8. This feature eliminates the need to retype the same list of variables each time the user wishes to examine the same group. Refer to the TYPE command for the proper format of the list.

The command format is

GROUP n list

where

n = the group number 1–8

list = a string of TYPE statements to be called in future accessing of the current group number.

GROUP

with no arguments will cause FORDDT to type out the current contents of all the groups.

GROUP n

will type out the contents of the particular group requested.

Note that one group may call another.

LOCATE

Lists the program unit names in which a given symbol is defined. This is useful when the variable the user wishes to locate is not in the currently open program unit and is defined in more than one program unit. The command format is

LOCATE n

where n may be any FORTRAN variable, array, label, line number, or FORMAT statement number.

MODE

Defines the default formats of typeout from FORDDT. In initial default mode variables will be typed in floating point format. If the user wishes to change the typeout modes, the command format is

MODE list

where list contains one or more of the modes in the following table. (Only the first character of each mode need be typed to identify it to FORDDT.)

Mode	Meaning
F	FLOATING POINT
D	DOUBLE PRECISION
C	COMPLEX
I	INTEGER
O	OCTAL
A	ASCII (left-justified)
R	RASCII (right-justified)

A typical command string might be

MODE A,I,OCTAL

NEXT

Allows the user to cause FORDDT to trace source lines, statement labels, and entry point names during execution of the user program. This command will only provide trace facilities if the program was compiled with the DECsystem-20 FORTRAN /DEBUG switch. If this switch was not used, the NEXT command will act as a CONTINUE command. The command format is

NEXT n/sw

where

n = a program variable or integer numeric value and
sw = one of the following switches

/S = statement label
/L = source line
/E = entry point

The default starting value of n is 1, a single statement trace. The default switch is /L.

The command

NEXT 20/L

will trace the execution of the next 20 source line numbers or until another pause is encountered.

Note that if no argument is specified, the last argument given will be used. For example

NEXT /E

will change the tracing mode to trace only subprogram entries using the numeric argument previously supplied.

OPEN

Allows the user to open a particular program unit of the loaded program so that the variables will be accessible to FORDDT. Any previously opened program unit is closed automatically when a new one is opened. Only global symbols, symbols in the currently open unit, and unique locals are available at any one time. Note that starting FORDDT automatically opens the MAIN program. The command format is

OPEN name

where name is the subprogram name. OPEN with no arguments will reopen the MAIN program.

If the PROGRAM statement was used in the FORTRAN program, the name supplied by the user will be requested upon entering FORDDT.

PAUSE

Allows the user to place a pause request at a statement number, source line number, or subroutine entry point. Up to 10 pauses may be set at any one time. When a pause is encountered, execution is suspended at that point and control is returned to FORDDT. Also, when a pause is encountered, the symbol table of that subprogram is automatically opened.

The command formats include

PAUSE P
PAUSE P AFTER n
PAUSE P IF condition
PAUSE P TYPING /g
PAUSE P AFTER n TYPING /g
PAUSE P IF condition TYPING /g

where

P = the point where the pause is requested,
n = an integer numeric constant
g = a group number

PAUSE 100

will set a pause at statement label 100, cause execution to be suspended, and cause FORDDT to be entered on reaching 100 in the program.

PAUSE #245 AFTER MAX(5)

will cause a pause to occur at source line number 245, after encountering this point the number of times specified by MAX(5). Note that AFTER may not be abbreviated.

PAUSE DELTA IF LIMIT(3.1).GT.2.5E-3

If the variable LIMIT(3.1) is greater than the value 2.5E-3, the pause request will be granted. The IF may not be abbreviated, but all the usual FORTRAN logical connectives are allowed.

PAUSE 505 TYPING /5

will request a pause to be made at the first occurrence of the label 505, and the variables in group 5 will be displayed. The TYPING specification may not be abbreviated.

PAUSE LINE#24 AFTER 16 TYPING 3

will place a request at source line number 24 after 16 (octal) times through, however, the contents of group 3 will be displayed every time.

When the TYPING option is used with the PAUSE command, control can be transferred to FORDDT at the next typeout by typing any character on the terminal.

Note that Pause requests remain after a CTRL/C REENTER sequence, a START command, or a CTRL/C START sequence.

REMOVE

Removes the previously requested pauses. The command format is

REMOVE P

For example

REMOVE L#123

will remove a pause at program source line number 123.

REMOVE ALPHA

will remove a pause at the subroutine entry to ALPHA.

REMOVE with no arguments will remove all the user's pause requests, and in this case, no abbreviation of REMOVE is allowed. This prevents the unintentional removal of pauses.

START

Starts the user program at the normal FORTRAN main program entry point. The command format is

START

STOP

Terminates the program, requests FOROTS to close all open files, and causes an exit to the monitor. The usual command format is

```
STOP
```

```
STOP/RETURN
```

will allow a return to monitor mode without releasing devices or closing files so that a CONTINUE can be issued.

STRACE

Displays a subprogram level backtrace of the current state of the program. The command format is

```
STRACE
```

TYPE

Causes one or more FORTRAN defined variables, arrays, or array elements to be displayed on the user terminal. The command format is

```
TYPE list
```

where list may be one or more variable or array references and/or group numbers. These specifications must be separated by commas, and group numbers must be preceded by a slash (/). The command with no arguments will use the last argument list submitted to FORDDT.

An array element range can also be specified, for example

```
TYPE PI(5)–PI(13)
```

will display the values from PI(5) to PI(13) inclusive. If an unsubscripted array name is specified, the entire array will be typed.

There are several methods of choosing the form of typeout in conjunction with the MODE command.

1. If the user does not specify a format, the default is floating point form.

2. The user can specify a format via the MODE command described in this appendix.

3. The user can change the format previously designated by the MODE command by including print modifiers in the TYPE or GROUP string. The print modifiers are

   ```
   /A, /C, /D, /F, /I, /O, /R
   ```

   The first print modifier specified in a string of variables determines the mode for the entire string unless another mode is placed directly to the right of a particular variable. For example in

   ```
   TYPE /I K, L/O, M, N/A, /2
   ```

the typeout mode is integer until another mode is specified. Therefore,

K, M, and /2 = Integer
L = OCTAL
N = ASCII

WHAT — Displays the information saved by FORDDT. The command format is

WHAT

F.7 ENVIRONMENT CONTROL

If a program enters an indefinite loop, the user can recover by typing a CTRL/C CTRL/C REENTER sequence. This action will cause FORDDT to simulate a pause at the point of reentry and allow the user to control his run-away program.

Most commands can be used once the program has been reentered; however, GOTO, STRACE, TYPE, and ACCEPT cause transfer of control to routines external to FORDDT. No guarantee can be made to assure that any of these commands following a CTRL/C CTRL/C REENTER sequence will not destroy the user profile. The program must be returned to a stable state before any of these four commands can be issued. In order to restore program integrity, the user should set a pause at the next label and then CONTINUE to it. If the /DEBUG:TRACE switch was used, a NEXT 1 command can be issued to restore program integrity.

F.8 FORTRAN/OPTIMIZE SWITCH

The user should never attempt to use FORDDT with a program that has been compiled with the /OPTIMIZE switch. The global optimizer causes variables to be kept in AC's. For this reason, attempts to examine or modify variables in optimized programs will not work. Also, since the optimizer moves statements around in the user program, attempts to trace program flow will lead to great confusion.

F.9 FORDDT MESSAGES

FORDDT responds with two levels of messages – fatal error and warning. Fatal error messages indicate that the processing of a given command has been terminated. Warning messages provide helpful information. The format of these messages is

?FDTXXX text
or
%FDTXXX text

where

?	=	fatal
%	=	warning
FDT	=	FORDDT mnemonic
XXX	=	3 letter mnemonic for error message
text	=	explanation of error

Square brackets ([ ]) in this section signify variables and are not output on the terminal.

Fatal Errors

The fatal errors in the following list are each preceded by ?FDT on the user terminal and on listings. They are listed in alphabetical order.

BDF [symbol] IS UNDEFINED OR IS MULTIPLY DEFINED

BOI BAD OCTAL OUTPUT

An illegal character was detected in an octal input value.

CCN CANNOT CONTINUE

Pause has been placed on some form of skip instruction causing FORDDT to loop; should never be encountered in FORTRAN compiled programs.

CFO CORE FILE OVERFLOW

The storage area for GROUP text has been exhausted.

CNU THE COMMAND [name] IS NOT UNIQUE

More letters of the command are required to distinguish it from the other commands.

CSH CANNOT 'START' HERE

The specified entry point is not an acceptable FORTRAN main program entry point.

DTO DIMENSION TABLE OVERFLOW

FORDDT does not have the space to record any more array dimensions until some are removed.

FCX FORMAT CAPACITY EXCEEDED

An attempt was made to specify a FORMAT statement requiring more space than was originally allocated by FORTRAN.

FNI FORMAL NOT INITIALIZED

Reference to a FORMAL parameter of some subprogram that was never executed.

FNR [array name] IS A FORMAL AND MAY NOT BE RE-DEFINED

FORMAL parameters may not be DIMENSIONed.

IAF ILLEGAL ARGUMENT FORMAT

The parameters to the given command were not specified properly. Please refer to the documentation for correct format.

IAT ILLEGAL ARGUMENT TYPE = [number]

An unrecognized subprogram argument type was detected. Please submit an SPR if this message occurs.

ICC COMPARE TWO CONSTANTS IS NOT ALLOWED

Conditional test involves two constants.

IER E

Internal FORDDT error – please report via an SPR.

IGN INVALID GROUP NUMBER

Group numbers must be integral and in the range 1 through 8.

INV INVALID VALUE

A syntax error was detected in the numeric parameter.

ITM ILLEGAL TYPE MODIFIER – S

The mode S is only valid for ACCEPT statements.

LGU [array name] LOWER SUBSCRIPT.GE.UPPER

The lower bound of any given dimension must be less than or equal to the upper bound.

LNF [label] IS NOT A FORMAT STATEMENT

MLD [array name] MULTI-LEVEL ARRAY DEFINITION NOT ALLOWED

The same array cannot be dimensioned more than once (via the [dimensions] construct) in a single command.

MSN MORE SUBSCRIPTS NEEDED

The array is defined to have more dimensions than were specified in the given reference.

NAL NOT ALLOWED

An attempt has been made to modify something other than data or a FORMAT.

NAR NOT AFTER A RE-ENTER

The given command is not allowed until program integrity has been restored via a CONTINUE or NEXT command.

NDT DDT NOT LOADED

NFS CANNOT FIND FORTRAN START ADDRESS FOR [program name]

Main program symbols are not loaded.

NFV [symbol] IS NOT A FORTRAN VARIABLE

Names must be six character alpahnumeric strings beginning with a letter.

NGF CANNOT GOTO A FORMAT STATEMENT

NPH CANNOT INSERT A PAUSE HERE

An attempt has been made to place a pause at other than an executable statement or subprogram entry point.

NSP [symbol] NO SUCH PAUSE

An attempt has been made to REMOVE a pause that was never set up.

NUD [symbol] NOT A USER DEFINED ARRAY

An attempt has been made to remove dimension information for an array that was never defined.

PAR PARENTHESES REQUIRED (. .)

Parentheses are required for the specification of FORMAT statements and complex constants.

PRO TOO MANY PAUSE REQUESTS

The PAUSE table has been exhausted. 10 is the maximum limit.

SER SUBSCRIPT ERROR

The subscript specified is outside the range of its defined dimensions.

STL [array name] SIZE TOO LARGE

An attempt has been made to define an array larger than 256K.

TMS TOO MANY SUBSCRIPTS

The array is defined to have less dimensions than are specified in the given element reference.

URC UNRECOGNIZED COMMAND

Warning Messages

Each warning message in this list is preceded by %FTN on the user terminal and on listings. They are given here in alphabetical order.

ABX [array name] COMPILED ARRAY BOUNDS EXCEEDED

FORDDT has detected another symbol defined in the specified range of the array. Note that this will occur in certain EQUIVALENCE cases and can be ignored at that time.

CHI CHARACTERS IGNORED: "[text]"

The portion of the command string included in ". . ." was thought to be extraneous and was ignored.

NAR [symbol] IS NOT AN ARRAY

NSL NO SYMBOLS LOADED

FORDDT cannot find the symbol table.

NST NOT 'STARTED'

The specified command requires that a START be previously issued to assure that the program is properly initialized.

POV PROGRAM OVERLAYED

The symbol table is different from the last time FORDDT had control.

SFA SUPERSEDES ARRAY

The FORTRAN generated dimension is being superseded for the given array.

SPO VARIABLE IS SINGLE PRECISION ONLY

XPA ATTEMPT TO EXCEED PROGRAM AREA WITH [symbol name]

An attempt has been made to access memory outside of the currently defined program space.

APPENDIX G
COMPILER MESSAGES

DECsystem-20 FORTRAN responds with two levels of messages – fatal error and warning. If a warning message is received the compilation will continue, but a fatal error will stop the program from being compiled. The format of messages is

```
?FTNXXX LINE:n text
        or
%FTNXXX LINE:n text
```

where

?	=	fatal
%	=	warning
FTN	=	FORTRAN mnemonic
XXX	=	3 letter mnemonic for the error message
LINE:n	=	line number where error occurred
text	=	explanation of error

Square brackets ([ ]) in this appendix signify variables and are not output on the terminal.

Fatal Errors

Each fatal error in the following list is preceded by ?FTN on the user terminal and on listings. They are presented here in alphabetical order.

ABD [symbolname] HAS ALREADY BEEN DEFINED [definition]

The usage given conflicts with current information about the symbol. For example, a symbol defined in an EQUIVALENCE statement cannot be referenced as a subprogram name.

AWN ARRAY REFERENCE [name] HAS WRONG NUMBER OF SUBSCRIPTS

The array was defined to have more or less dimensions than the given reference.

BOV STATEMENT TOO LARGE TO CLASSIFY

To determine statement type, some portion of the statement must be examined by the compiler before actual semantic and syntax analysis begins. During this classification the entire portion of the statement required must fit into the internal statement buffer (large enough for a normal 20 line statement). This error message is issued when the portion of a given statement required for classification is too large to fit in the buffer. Once FORTRAN has classified a statement, there is no explicit restriction on its length.

CER COMPILER ERROR IN ROUTINE [name]
Please submit an SPR for any occurrence of this message.

CFF CANNOT FIND FILE
The file referenced in an INCLUDE statement was not found.

CPE CHECKSUM OR PARITY ERROR ON [source/listing/object] FILE [name]

CQL NO CLOSING QUOTE IN LITERAL

CSF ILLEGAL STATEMENT FUNCTION REFERENCE IN CALL STATEMENT

DDA [symbolname] IS DUPLICATE DUMMY ARGUMENT

DFC VARIABLE DIMENSION [name] MUST BE SCALAR DEFINED AS FORMAL OR IN COMMON

DFD DOUBLE [type] NAME ILLEGAL
Duplicate fields were encountered in an INCLUDE file specification.

DIA DO INDEX VARIABLE [name] IS ALREADY ACTIVE
In any nest of DO loops, a given index variable may not be defined for more than one loop.

DID CANNOT INITIALIZE A DUMMY PARAMETER IN DATA

DLN OPTIONAL DATA VALUE LIST NOT SUPPORTED
The extended FORTRAN statement form that allows data values to be defined in type specification statements is not supported by FORTRAN.

DSF ARGUMENT [name] IS SAME AS FUNCTION NAME

DTI THE DIMENSIONS OF [arrayname] MUST BE OF THE TYPE INTEGER

DVE CANNOT USE DUMMY VARIABLES IN EQUIVALENCE

DWL [source/listing/object] DEVICE [[device]] WRITE LOCKED

ECT ATTEMPT TO ENTER [symbolname] INTO COMMON TWICE

EDN EXPRESSION TOO DEEPLY NESTED TO COMPILE

EID ENTRY STATEMENT ILLEGAL INSIDE A DO LOOP

EIM ENTRY STATEMENT ILLEGAL IN MAIN PROGRAM

ENF LABEL [number] MUST REFER TO AN EXECUTABLE STATEMENT, NOT A FORMAT

ETF ENTER FAILURE [filename]

EXB EQUIVALENCE EXTENDS COMMON BLOCK [name] BACKWARD

FEE FOUND [symbol] WHEN EXPECTING [symbol] OR A [symbol]
General syntax error message.

FNE LABEL [number] MUST REFER TO A FORMAT, NOT AN EXECUTABLE STATEMENT

FWE FOUND [symbol] WHEN EXPECTING A [symbol]

HDE HARDWARE DEVICE ERROR ON [source/listing/object] DEVICE [[device]]

IAC ILLEGAL ASCII CHARACTER [character] IN LABEL FIELD

IAL INCORRECT ARGUMENT TYPE FOR LIBRARY FUNCTION [name]

IBK ILLEGAL STATEMENT IN BLOCKDATA SUBPROGRAM

ICL ILLEGAL CHARACTER [character] IN LABEL FIELD

IDN DO LOOP AT LINE: [number] IS ILLEGALLY NESTED

The user is attempting to terminate a DO loop before terminating one or more loops defined after the given one.

IDS IMPLICIT DO INDICES MAY NOT BE SUBSCRIPTED

IDT ILLEGAL OR MISSPELLED DATA TYPE

IDV IMPLIED DO INDEX IS NOT VARIABLE

IED INCONSISTENT EQUIVALENCE DECLARATION

The given EQUIVALENCE declaration would cause some symbolic name to refer to more than one physical location.

IFD INCLUDED FILES MUST RESIDE ON DISK

IID NON-INTEGER IMPLIED DO INDEX

IIP ILLEGAL IMPLICIT SPECIFICATION PARAMETER

IIS INCORRECT INCLUDE SWITCH

ILF ILLEGAL STATEMENT AFTER LOGICAL IF

Refer to section 9.3.2 for restrictions on logical IF object statements.

INN INCLUDE STATEMENTS MAY NOT BE NESTED

IOD ILLEGAL STATEMENT USED AS OBJECT OF DO

ISD ILLEGAL SUBSCRIPT EXPRESSION IN DATA STATEMENT

Subscript expressions may be formed only with implicit DO indices and constants combined with +, -, *, or /.

ISN [symbolname] IS NOT [symboltype]

The symbol cannot be used in the attempted manner.

IUT PROGRAM UNITS MAY NOT BE TERMINATED WITHIN INCLUDED FILES

IVP INVALID PPN

IXM ILLEGAL MIXED MODE ARITHMETIC
Complex and double precision cannot appear in the same expression.

IZM ILLEGAL [datatype] SIZE MODIFIER [number]
Refer to section 6.3.

LAD LABEL [number] ALREADY DEFINED AT LINE [number]

LED ILLEGAL LIST DIRECTED [statement type]

LFA LABEL ARGUMENTS ILLEGAL IN FUNCTION OR ARRAY REFERENCE

LGB LOWER BOUND GREATER UPPER BOUND FOR ARRAY [name]

LLS LABEL TOO LARGE OR TOO SMALL
Labels cannot be 0 or greater than 5 digits.

LNI LIST DIRECTED I/O WITH NO I/O LIST

LTL TOO MANY ITEMS IN LIST – REDUCE NUMBER OF ITEMS
In certain rare instances, a combination of long lists in a single statement can exhaust the syntax stack.

MCE MORE THAN 1 COMMON VARIABLE IN EQUIVALENCE GROUP

MSP STATEMENT NAME MISSPELLED

MWL ATTEMPT TO DEFINE MULTIPLE RETURN WITHOUT FORMAL LABEL ARGUMENTS

NCF NOT ENOUGH CORE FOR FILE SPECS. TOTAL K NEEDED = [number]

NEX NO EXPONENT AFTER D OR E CONSTANT

NFS NO FILENAME SPECIFIED
The INCLUDE statement requires a filename.

NGS CANNOT GET SEGMENT [name] – ERROR CODE: 0
Error code 0 indicates that the file cannot be found.

NIR REPEAT COUNT MUST BE AN UNSIGNED INTEGER

NIU NON-INTEGER UNIT IN I/O STATEMENT

NLF WRONG NUMBER OF ARGUMENTS FOR LIBRARY FUNCTION [name]

NNF NO STATEMENT NUMBER ON FORMAT

NRC STATEMENT NOT RECOGNIZED

NUO	.NOT. IS A UNARY OPERATOR
NWD	INCORRECT USE OF * OR ? IN [filename]
OPW	OPEN PARAMETER [name] IS OF WRONG TYPE
PIC	THE DO PARAMETERS OF [index name] MUST BE INTEGER CONSTANTS
PRF	PROTECTION FAILURE [filename]
QEF	QUOTA EXCEEDED OR DISK FULL [filename]
QEX	BLOCK TOO LARGE OR QUOTA EXCEEDED FOR [source/listing/object] FILE [name]
RDE	RIB OR DIRECTORY ERROR [filename]
RFC	[function name] IS A RECURSIVE FUNCTION CALL
RIC	COMPLEX CONSTANT CANNOT BE USED TO REPRESENT THE REAL OR IMAGINARY PART OF A COMPLEX CONSTANT
SAD	ARRAY [name] – SIGNED DIMENSIONS MAY APPEAR ONLY AS CONSTANT RANGE LIMITS
SNL	[statement name] STATEMENTS MAY NOT BE LABELLED
SOR	SUBSCRIPT OUT OF RANGE
TFL	TOO MANY FORMAT LABELS SPECIFIED
TOF	MORE THAN 2 OUTPUT FILES ARE NOT ALLOWED Only a listing and a relocatable binary file may be specified as output files.
UMP	UNMATCHED PARENTHESES
USI	[symbol type] [symbol name] USED INCORRECTLY The given symbol cannot be used in this way.
VNA	SUBSCRIPTED VARIABLE IN EQUIVALENCE BUT NOT AN ARRAY
VSE	EQUIVALENCE SUBSCRIPTS MUST BE INTEGER CONSTANTS

Warning Messages

Each warning message in the following list is preceded by %FTN on the user terminal and on listings. They are presented here in alphabetical order.

AGA	OPT–OBJECT VARIABLE, OF ASSIGNED GOTO WITHOUT OPTIONAL LIST, WAS NEVER ASSIGNED
CAI	COMPLEX EXPRESSION USED IN ARITHMETIC IF
CTR	COMPLEX TERMS USED IN A RELATIONAL OTHER THAN EQ OR NE The result of the other relational operators with complex operands is undefined.

CUO CONSTANT UNDERFLOW OR OVERFLOW

This message is issued when overflow or underflow is detected as the result of building constants or evaluating constant expressions at compile time.

DIM DO INDEX MODIFIED INSIDE LOOP

A program which does this may be incorrectly compiled by the optimizer since it assumes that indices are never modified. Note that the number of iterations is calculated at the beginning of the loop and is never affected by modification of the index within the loop.

DIS PROGRAM IS DISCONNECTED – OPTIMIZATION DISCONTINUED

Please submit an SPR if this message occurs.

FMR MULTIPLE RETURNS DEFINED IN A FUNCTION

FNA A FUNCTION WITHOUT AN ARGUMENT LIST

ICC ILLEGAL CHARACTER CONTINUATION FIELD OF INITIAL LINE

Continuation lines cannot follow comment lines.

ICD INACCESSIBLE CODE, STATEMENT DELETED

The optimizer will delete statements which cannot be reached during execution.

ICS ILLEGAL CHARACTER IN LINE SEQ#

IFL INFINITE LOOP, OPTIMIZATION DISCONTINUED

LID IDENTIFIER [name] MORE THAN 6 CHARACTERS

The remaining characters are ignored.

MVC NUMBER OF VARIABLES DOES NOT EQUAL THE NUMBERS OF CONSTANTS IN DATA STATEMENT

NED NO END STATEMENT IN PROGRAM

NOD GLOBAL OPTIMIZATION NOT SUPPORTED WITH /DEBUG – /OPT IGNORED

NOF NO OUTPUT FILES GIVEN

PPS PROGRAM STATEMENT PARAMETERS IGNORED

For compatibility purposes.

RDI ATTEMPT TO REDECLARE IMPLICIT TYPE

SOD [name] STATEMENT OUT OF ORDER

VND FUNCTION RETURN VALUE IS NEVER DEFINED

VNI VARIABLE [name] IS NOT INITIALIZED

The optimizer analysis determined that the given variable was never initialized prior to its use in a calculation.

WOP OPT–WARNINGS GIVEN IN PHASE 1, OPTIMIZED CODE MAY NOT BE CORRECT

One or more of the messages issued prior to this message resulted from situations which violate assumptions made by the optimizer and thus may cause it to generate code that does not execute as desired.

XCR EXTRANEOUS CARRIAGE RETURN

Carriage return was not immediately preceded or followed by a line termination character.

ZMT SIZE MODIFIER [number] TREATED AS [data type]

Message is issued when one of the data type size modifiers is used which is accepted only for compatibility.

APPENDIX H
DECSYSTEM-10 COMPATIBILITY

The following items are included in the DECsystem-20 FORTRAN software for compatibility with the DECsystem-10. They are not supported on the DECsystem-20. Users must not specify these items because their actions are undefined and the results cannot be guaranteed.

1. Logical Device Assignments.
   (Refer to pages 10-4 and E-27.)

Device	Logical unit number	Use
PTR	*06*	*Paper Tape Reader*
PTP	*07*	*Paper Tape Punch*
DIS	*08*	*Display*
DTA1	*09*	*DECtape*
DTA2	*10*	
DTA3	*11*	
DTA4	*12*	
DTA5	*13*	
DTA6	*14*	
DTA7	*15*	*DECtape*

2. PUNCH Statement

3. KA10 and KI10 compiler switches

4. The following Library Subroutines:

   AXIS
   LINE
   MKTBL
   NUMBER
   PLOT
   PLOTS
   SCALE
   SLITE(i)
   SLITET(i,j)
   SSWTCH(i,j)
   SYMBOL
   WHERE

5. DDT command to FORDDT.

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

DECsystem-20 FORTRAN
Reference Manual
DEC-20-LFRMA-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please cut along this line.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

If you require a written reply, please check here. ☐

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P.O. Box F
Maynard, Massachusetts 01754


digital