FORTRAN Reference Manual

Order No. AA-4158B-TM

April 1977

This document describes the language elements of the FORTRAN-20 compiler for the DECSYSTEM-20.

This document supersedes the document of the same name, Order No. DEC-20-LFMRA-A-D, published January 1976.

OPERATING SYSTEM AND VERSION:	Any Digital-supported operating system for the DECSYSTEM-20.
SOFTWARE VERSION:	FORTRAN-20, Version 5

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, January 1976
Revised: April 1977

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.

5/77-15

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

TABLES

CONTENTS (CONT.)

CHAPTER 1

PROLOGUE

1.1 BACKGROUND

A FORTRAN source program consists of statements constructed using the language elements and the syntax described in this manual. A statement performs one of the following functions:

1. Causes operations such as multiplication, division, and branching to be carried out.

2. Specifies the type and format of data being processed.

3. Specifies the characteristics of the source program.

FORTRAN statements are composed of keywords, i.e., words that 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 12 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, (constants, variables, and expressions), the character set from which they may be formed, and the rules that govern their construction and use are described in Chapters 2 through 4.

Table 1-1
FORTRAN Statement Categories

Chapter Reference	Category Name	Function
5	Compilation Control Statements	Identify programs and indicate their beginning and ending points.
6	Specification Statements	Declare the properties of variables, arrays, and functions.
7	DATA Statements	Assign initial values to variables and array elements.
8	Assignment Statements	Assign values to named variables and array elements.
9	Control Statements	Determine the order of execution of the object program and terminate its execution.
10	Input/Output Statements	Transfer data between internal storage and specified input/output devices.
11	NAMELIST Statements	Establish lists that are used with certain input/output statements to transfer data that appears in a special type of record.
12	File Control Statements	Identify, open, and close files and set parameters for input and output operations between files and the processor.
13	FORMAT Statements	Specify formats for data on input/output devices.
14	Device Control Statements	Control the positioning of records or files on certain input/output devices.
15	Subprogram Statements	Define functions and subroutines and their entry points.
16	BLOCK DATA Statements	Define data specification subprograms that may initialize common storage areas.

PREFACE

This manual describes the FORTRAN language as implemented for the DECsystem-20 FORTRAN Language Processing System. In the text, the language is called FORTRAN-20 (to distinguish it from ANSI FORTRAN), or simply FORTRAN.

Since this is a reference manual, we assume that you have used FORTRAN before. If you haven't, you should read one of the many introductory FORTRAN texts.

Your use of FORTRAN may also require use of other DECsystem-20 programs: the monitor, the CREF program, the debugging program, a text editor, and the BATCH program. These are described in the following manuals:

User's Guide
DEC-20-OUGAA-A-D

Monitor Calls User's Guide
DEC-20-UMUGA-A-D

EDIT User's Guide
DEC-20-UEUGA-A-D

BATCH Reference Manual
DEC-20-OBRMA-A-D

The standard for FORTRAN is the American National Standards Institute (ANSI) FORTRAN, X3.9-1966. FORTRAN-20 extensions and additions to ANSI FORTRAN are gray shaded.

CHAPTER 2

CHARACTERS AND LINES

2.1 CHARACTER SET

Table 2-1 lists the digits, letters, and symbols recognized by FORTRAN. The remainder of the ASCII-1968 character set(1), is acceptable within literal constants or comment text, but these characters cause fatal errors in other contexts. An exception is CONTROL-Z, which, when used in terminal input, means end-of-file.

NOTE

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

Table 2-1
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	

Digits	
0	5
1	6
2	7
3	8
4	9

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.

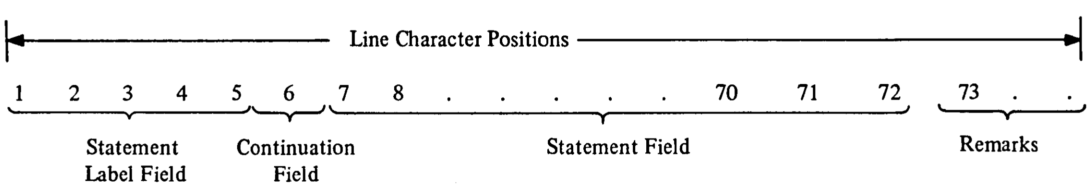
Table 2-1 (Cont.)
FORTRAN Character Set

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 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

You may place a number ranging from 1 to 99999 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. Leading zeros and all blanks in the label field are ignored, e.g., the numbers 00105 and 105 are both accepted as statement number 105. You may assign the statement numbers in a source program in any order; however, each statement number must be unique with respect to all other statements in the program or subprogram. You cannot label non-executable statements other than FORMAT and END statements.

A main program and a subroutine may contain identical statement numbers. In this case, references to these numbers are understood to mean the numbers in the same program unit in which the reference is made. An example:

> Assume that main module MAINMD and subprogram SUB1 both contain statement number 105. A GO TO statement, for instance, in MAINMD will refer to statement 105 in MAINMD, NOT to 105 in SUB1. A GO TO in SUB1 will transfer control to 105 in SUB1.

When you enter source programs into the system via a standard user terminal, you may use an initial tab 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 Section 2.3.1 for description.)

Whenever you use a tab to skip all or part of the label field of a continuation line, the next character you enter 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 consisting 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, usually 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 Section 2.3.3.)

2.3 LINE TYPES

A line in a FORTRAN source program may be:

1. An initial line,
2. A continuation line,
3. A multi-statement line,
4. A comment line,
5. A debug line, or
6. A blank line.

Each of these 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.

An initial line may be assigned a statement number and must have either a blank or a zero in its continuation line field, i.e., character position 6.

If you enter an initial line via a keyboard input device, you may use an initial tab to skip all or part of the label field. If you use an initial tab for this purpose, you must immediately follow it with a non-numeric character, i.e., the first character of the statement field must be non-numeric.

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 you are entering a continuation line via a keyboard, you may use an initial tab 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 4-line FORTRAN FORMAT statement using initial tabs:

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

        Continuation Line Characters, i.e., 2, 3, and 4
```

2.3.2 Multi-Statement Lines

You may write more than one FORTRAN statement in the statement field of one line. The rules for structuring a multi-statement line are:

1. Successive statements must be separated by semicolons (;).
2. Only the first statement in the series can have a statement number.
3. Statements following the first statement cannot be a continuation of the preceding statement.
4. The last statement in a line may be continued to the next line if that next 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.

To structure a comment line:

1. You must place one of the characters C (or c), $,/,*, or ! in character position 1 of the line to identify it as a comment line.
2. You may write the text into character positions 2 through the end of the line.
3. You may place comment lines anywhere in the source program, but they cannot precede a continuation line because comments terminate a continuation sequence.
4. You may write a large comment as a sequence of any number of lines; however, each line must carry the identifying character (C,$,/,*, 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.

You may add a remark 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 Section 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 is present in the command string. (Refer to Appendix C for a description of the file 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

You may insert lines consisting of only blanks, tabs, or no characters anywhere in your 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 source 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 or BASIC. These sequence numbers are used in place of the listing line numbers normally generated.

2.4 ORDERING OF STATEMENTS

The order in which you place statements in a program unit is important. That is, certain types of statements have to be processed before others to guarantee that compilation takes place as you expect. The proper sequence for 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="4">END Statement</td></tr>
</table>

Horizontal lines indicate the order in which statements must appear. That is, you cannot intersperse horizontal sections. 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, you may intersperse DATA statements with statement function definitions and executable statements. you may intersperse FORMAT statements with IMPLICIT statements, parameter statements, other specification statements, DATA statements, statement function definitions, and executable statements. The only restriction on the placement of FORMAT statements is that they must appear after any PROGRAM, FUNCTION, subprogram, and BLOCK DATA statements, and before the END statement.

Special Cases:

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

2. 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.

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

When statements are out of place, FORTRAN issues messages, some of which may indicate fatal errors.

CHAPTER 3

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

3.1 DATA TYPES

The data types you may use in FORTRAN source programs are:

1. integer,
2. real,
3. double-precision,
4. complex,
5. octal,
6. double-octal,
7. literal,
8. statement label, and
9. 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 (Sections 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 you may use in 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 statement label

3.2.1 Integer Constants

An integer constant is a string of from one to eleven digits that represents a whole decimal number (a number without a fractional part). Integer constants must be within the range of (-2**35)-1 to (+2**35)-1 (-34359738367 to +34359738367). Positive integer constants may optionally be signed; negative integer constants must be signed. You cannot use decimal points, commas, or other symbols 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:

1. A basic real constant: a string of decimal digits followed immediately by a decimal point followed optionally by a decimal fraction, e.g., 1557.42.

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

3. 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 (7 to 9 decimal digits).

Precision for real constants is maintained to approximately eight significant digits; the absolute precision depends upon the numbers involved.

The exponent field of a real constant written in E notation form cannot be empty (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., $-38 < n \leq +38$). The following are examples of valid real constants.

```
-98.765
7.0E+0    (7.)
.7E-3     (.0007)
5E+5      (500000.)
50115.
50.E1     (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:

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

2. Each double-precision constant occupies two storage locations.

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

You must use both the letter D and an exponent (even of zero) in writing a double-precision constant. The exponent need only be signed if it is negative; its magnitude must be greater than -38 and equal to or less than +38 (i.e., $-38 < n \leq +38$). The range of magnitude permitted a double-precision constant is 0.14 X 10**(-38) to 3.4 X 10**(+38)

The following are examples of valid double-precision constants.

```
7.9D03     (= 7900)
7.9D+03    (= 7900)
7.9D-3     (= .0079)
79D03      (= 79000)
79D0       (= 79)
```

The following are examples of invalid double-precision constants.

```
7.9D99     (exponent is too large)
7.9E5      ("E" denotes a single-precision constant)
```

3.2.4 Complex Constants

You can represent a complex constant 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

You may use octal numbers (radix 8) 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 zeros to fill their storage location.

If you specify more than 12 digits 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; zeros are added to fill any unused digits.

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

All octal constants must:

1. be preceded by a double quote (") to identify the digits as octal, e.g., "777, and

2. be signed if negative, but optionally signed if positive.

3. contain one or more of the digits 0 through 7, but not 8 or 9.

The following are examples of valid octal constants:

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

The following are examples of invalid octal constants:

```
"12368     (contains an 8)
7777       (no identifying double quotes)
```

When you use an octal constant as an operand in an expression, its form (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.

Octal constants may not be used as stand-alone arguments for library functions that require non-octal arguments. MIN0, for instance, requires INTEGER arguments and cannot accept octal arguments.

When you combine a double octal constant 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.. Always write logical constants 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:

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

2. 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 that 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.

You may enter literal constants into DATA statements as a string of:

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

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

The 7-bit ASCII characters that 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 as 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.

You write statement label constants as strings 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 a statement label constant.

You use statement label constants only in the argument list of CALL statements to identify 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. You may use more than six characters, but FORTRAN will ignore 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)
```

You use symbolic names to identify specific items of a source program; Table 3-2 lists these items, together with an example of a symbolic name and text reference for each.

Table 3-2
Use of Symbolic Names

Symbolic Names Can Identify	For Example	For a Detailed Description See Section
1. Variables	PI, CONST, LIMIT	3.4
2. Arrays	TAX	3.5
3. Array elements	TAX (NAME, INCOME)	3.5.1
4. Functions	MYFUNC, VALFUN	15.2
5. Subroutines	CALCSB, SUB2, LOOKUP	15.5
6. External library functions	SIN, ATAN, COSH	15.4
7. COMMON block names	DATAR, COMDAT	6.5

3.4 VARIABLES

A variable is a datum (storage location) that is identified by a symbolic name and is not a constant, an array or an array element. Variables specify values that 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 you assign a value to a variable, it is termed an undefined variable, and you should not reference it except to assign a value to it.

If you reference an undefined variable, an unknown value (garbage) will be obtained.

The value you assign to a variable may be either a constant or the result of a calculation that 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 that it identifies. Variables may be:

1. integer
2. real
3. logical
4. double-precision, or
5. complex.

You may declare the type of a variable by using either implicit or explicit type declaration statements (Chapter 6). However, if you do not use type declaration statements, FORTRAN assumes the following convention:

1. Variable names that begin with the letters I, J, K, L, M, or N are normally integer variables.
2. Variable names that begin with any letter other than I, J, K, L, M, or N are normally 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 Section 3.3 for writing symbolic names.

Each datum within an array is called an array element. As with variables, you may assign a value to an array element. Before you assign a value to an array element it is considered to be undefined; you should not reference it until you have assigned it a value. If you reference an undefined array element, the value of the element will be unpredictable.

Name each element of an array 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

Give the subscript of an array element identifier within parentheses, as either one subscript quantity or a set of subscript quantities delimited by commas. Write the parenthesized subscript 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. You may use any number of subscript quantities in an element name; however, the number used must always equal the number of dimensions (Section 3.5.2) specified for the array.

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

1. 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.

2. 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.

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

4. 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.

Here are examples of valid array element subscripts:

1. IAB(1,5,3)

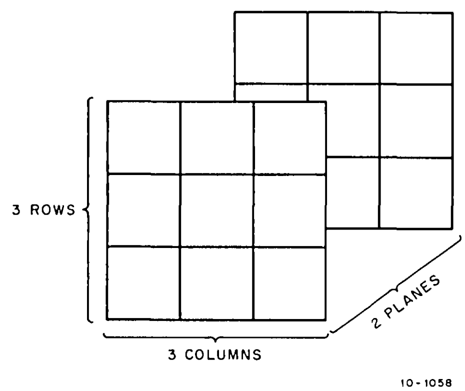
2. ABLE(A)

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

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

3.5.2 Dimensioning Arrays

You must declare the size (number of elements) of an array in order 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.



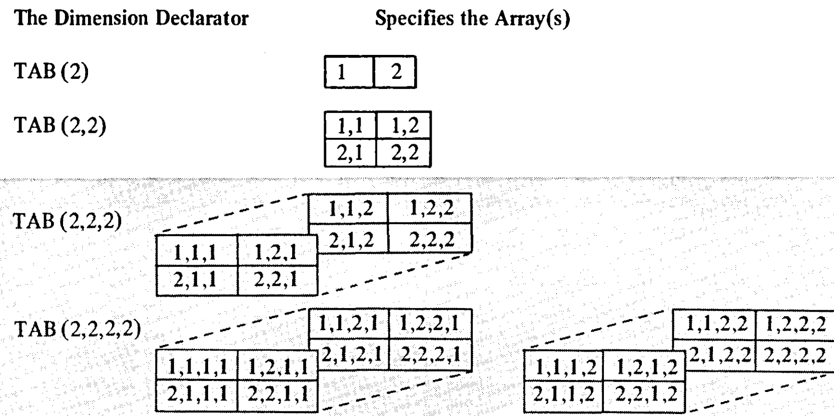
You specify the size of an array by an array declarator written as a subscripted array name. In an array declarator, however, each subscript quantity is a dimension of the array and must be either an integer variable or an integer constant.

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

The total number of elements that 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).

Dimension arrays 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, or plane) that it represents. The first three subscript positions specify the number of rows, columns, and planes that 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

FORTRAN-20 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. The value of the first subscript quantity varies between its minimum and maximum values most rapidly. The value of the last given subscript quantity increases 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)

In the following list, the elements of the three-dimensional array B(3,3,3) are stored row by row from left to right and from top to bottom.

B (1,1,1)	B (2,1,1)	B (3,1,1)
B (1,2,1)	B (2,2,1)	B (3,2,1)
B (1,3,1)	B (2,3,1)	B (3,3,1)
B (1,1,2)	B (2,1,2)	B (3,1,2)
B (1,2,2)	B (2,2,2)	B (3,2,2)
B (1,3,2)	B (2,3,2)	B (3,3,2)
B (1,1,3)	B (2,1,3)	B (3,1,3)
B (1,2,3)	B (2,2,3)	B (3,2,3)
B (1,3,3)	B (2,3,3)	B (3,3,3)

Thus B(3,1,1) is stored before B(1,2,1), and so forth.

CHAPTER 4

EXPRESSIONS

4.1 ARITHMETIC EXPRESSIONS

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

1. a constant
2. a variable
3. an array element
4. a function reference (see Chapter 14 for description), or
5. an arithmetic or logical expression written within parentheses.

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

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 parenthetical expression)

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

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

Observe the following rules in structuring compound arithmetic expressions:

1. 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 FORTRAN-20.

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

3. 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.

4. When you use 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 combinations and the type of the result of each combination are given in Table 4-3.

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

Type of Argument 2

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

Table 4-3
Permitted Base/Exponent Type Combinations

Base Operand	Exponent Operand			
	Integer	Real	Double Precision	Complex
Integer	Integer	Real	Double Precision	Complex
Real	Real	Real	Double Precision	Complex
Double Precision	Double Precision	Double Precision	Double Precision	(Prohibited)
Complex	Complex	Complex	(Prohibited)	Complex

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:

1. constant
2. variable
3. array element
4. function reference (see Chapter 15), or
5. another expression written within parentheses.

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

Table 4-4 gives the logical operators and a description of the operation each provides.

Table 4-4
Logical Operators

Operator	Description
.AND.	AND operator. Both of the logical operands combined by this operator must be true to produce a true result.
.OR.	Inclusive OR operator. If either or both of the logical operands combined by .OR. are true, the result will be true.
.XOR.	Exclusive OR operator. If either but not both of the logical operands combined by .XOR. is true, the result will be true.
.EQV.	Equivalence operator. If the logical operands being combined by .EQV. are both the same (both are true or both are false), the result will be true.
.NOT.	Complementation operator. This operator is used as a prefix that specifies complementation (inversion) of the item (operand or expression) that it modifies. The original item, if true by itself, becomes false, and vice versa.

Write logical expressions in the general form P .OP. Q, where P and Q are logical operand and .OP. is any logical operator but ".NOT.". The .NOT. operator complements the value of a logical operand; you must write it immediately before the operand that it modifies, e.g., .NOT.P. Table 4-5 is a truth table illustrating all possible logical combinations of two logical operands (P and Q) and the resultant of each combination.

When an operand of a logical expression is double-precision or complex, only the high-order word of the operand is used in the specified logical operation.

The assignment of a .TRUE. or a .FALSE. value to a given operand is based only on the sign of the numeric representation of the operand.

Table 4-5
Logical Operations, Truth Table

When P is	And Q is:	Then the Expression:	Is:
True	-----	.NOT.P	False
False	-----	.NOT.P	True
True	True	P .AND. Q	True
True	False	P .AND. Q	False
False	True	P .AND. Q	False
False	False	P .AND. Q	False
True	True	P .OR. Q	True
True	False	P .OR. Q	True
False	True	P .OR. Q	True
False	False	P .OR. Q	False
True	True	P .XOR. Q	False
True	False	P .XOR. Q	True
False	True	P .XOR. Q	True
False	False	P .XOR. Q	False
True	True	P .EQV. Q	True
True	False	P .EQV. Q	False
False	True	P. EQV. Q	False
False	False	P .EQV. Q	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 consisting of the foregoing variables are:

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

Logical functions are performed 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 are:

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 4-5 also illustrates all possible logical combinations of two one-bit binary operands (P and Q) and gives the result of each combination. Just read 1 for true and 0 for false.

4.2.1 Relational Expressions

Relational expressions consist 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-20, you may write relational operators either as a 2-letter mnemonic enclosed within periods, e.g., .GT., or symbolically using the symbols, >, <, = and #. Table 4-6 lists both the mnemonic and symbolic forms of the relational operators and specifies the type of quantitative test performed by each operator.

Table 4-6
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

Write relational expressions in the general form A(1) .OP.A(2), where A represents an arithmetic operand and .OP. is a relational operator.

You may mix arithmetic operands of type integer, real, and double precision in relational expressions.

You may compare complex operands 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 consisting of the foregoing variables are:

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

Examples of invalid relational expressions consisting 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 that use both logical and relational operators 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 following determine the order of computation of an expression:

1. the use of parentheses
2. an established hierarchy for the execution of arithmetic, relational, and logical operations and
3. the location of operators within an expression.

4.3.1 Parenthetical Subexpressions

In an expression, all subexpressions written within parentheses are evaluated first. When parenthetical 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 parenthetical expression is computed. When more than one operator is contained by a parenthetical subexpression, the required computations are performed according to the hierarchy of assigned operators (Section 4.3.2).

Example:

The separate computations performed in evaluating the expression

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

1. R1=A/B
2. 2=R1+C
3. R3=B/R2
4. R4=R3-C
5. R5=A+R4

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

4.3.2 Hierarchy of Operators

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

first, arithmetic operators,
second, relational operators, and
third, logical operators.

Table 4-7 specifies the precedence assigned to the individual operators of the foregoing classes.

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 4-7
Hierarchy of Operators

Class	Level	Symbol or Mnemonic
EXPONENTIAL	First	**
ARITHMETIC	Second Third Fourth	-(unary minus) and + (unary plus) *,/ +,-
RELATIONAL	Fifth	.GT.,.GE.,.LT.,.LE.,.EQ.,.NE. or >,>=,<,<=,==,#
LOGICAL	Sixth Seventh Eighth Ninth	.NOT. .AND. .OR. .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. But this left-to-right evaluation process can be overridden by parentheses. I/J*K(evaluated as(I/J) *K) does not equal I/(J*K),which is evaluated as written here.

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

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

Similarly, here too, parentheses alter the evaluation of the expression . (A**2)**B is evaluated in these two steps:
first R1 = A**2 (intermediate result)
second R2 = R1**2 (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:

1.R1 = I/J   R1 is integer

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

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

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

4.3.4 Use of Logical Operands in Mixed Mode Expressions

When you use logical operands 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

You use compilation control statements to identify 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 you to give the main program a name other than the compiler-assumed name "MAIN.". The general form of a PROGRAM statement is:

```
PROGRAM name
```

where:

name — is a 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 statements.)

5.3 INCLUDE STATEMENT

This statement allows you 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.ext[proj,prog]/NOLIST'
```

where:

dev: — is a device name. When no device name is specified, DSK: is assumed.

filename.ext — is the filename and extension of the statements that you wish to include. The name of the file is required; the extension is optional. If you specify "filename" only, .FOR is the assumed extension. If you specify the filename and period (filename.), the null extension is assumed. You may not specify wild (*) information.

[proj,prog] — is the project-programmer number. Your project-programmer number is assumed if none is specified. You cannot specify subdirectory information.

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

The following rules govern the use of the INCLUDE statement:

1. The INCLUDEd file may contain any legal 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.

2. 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 FORTRAN statements.)

3. The file(s) to be INCLUDEd must be on disk.

Note that an asterisk (*) is appended to the line numbers of the INCLUDEd statements on the compilation listing, provided /NOLIST is not specified.

5.4 END STATEMENT

Use this statement to show the physical end of a source program or subprogram. END is a nonexecutable statement. The general form of an END statement is:

```
END
```

The following rules govern the use of the END statement:

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

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

3. You may label an END statement.

CHAPTER 6

SPECIFICATION STATEMENTS

6.1 INTRODUCTION

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

1. DIMENSION
2. Statements that explicitly specify type, such as REAL or INTEGER
3. IMPLICIT
4. COMMON
5. EQUIVALENCE
6. EXTERNAL
7. PARAMETER

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

6.2 DIMENSION STATEMENT

DIMENSION statements identify and allocate the space required for source program arrays. You may specify any number of subscripted array names 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(max,...,max)   or   name(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 delimiters may be omitted from a dimension subscript. This is because minimum values are assumed to be 1 in the first place.

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 you may use a slash 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:

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

2. 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.

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

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

5. 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 SB1, SB2, or SB3 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 (the last value passed).

6.3 TYPE SPECIFICATION STATEMENTS

Type specification statements declare explicitly the data type of variable, array, or function symbolic names. You may give an array name 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.

Write type specification statements in the following form:

    type list

where type may be any one of the following declarators:

1. INTEGER
2. REAL
3. DOUBLE PRECISION
4. COMPLEX
5. LOGICAL

NOTE

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

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

In addition, you may append the data type size modifier 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 (one full word) real, and B and C are double-precision (two full words) real.

The list consists of any number of variable, array, or function names that 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 the following conventions:

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

2. Variable names, array names, and function names that 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 predefined FORTRAN-20 function name appears in a conflicting type statement, it is assumed that the name refers to a user-defined routine of the given type. If you place a generic function name in an explicit type statement, it loses 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 consists 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 parenthetical 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) starting 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 that 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 FORTRAN supplied functions.

You may specify a range of letters within the alphabet 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.

You may use more than one IMPLICIT statement, but they must appear before any other declaration statement in the program unit. Refer to Section 2.4 for a discussion on ordering FORTRAN statements.

6.5 COMMON STATEMENT

The COMMON statement enables you to establish storage that may be shared by two or more programs and/or subprograms and to name the variables and arrays that 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. Write COMMON statements in the following form:

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

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

The list (e.g., 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.

Either label COMMON storage areas or leave them blank (unlabeled). If the common area is to be labeled, give a symbolic name within slashes immediately before the list of items that is 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 that have the same name.

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

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

establishes one labeled (AREA1) and one unlabeled common area. Unlabeled common area is also called "blank common".

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 arrays, 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 variables and arrays as the reference 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, FORTRAN will string together all items listed for each labeled common area and for blank common areas 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
```

has 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 that contains the COMMON statement but not both.

Example

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

6.6 EQUIVALENCE STATEMENT

The EQUIVALENCE statement enables you 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. Write the EQUIVALENCE statement in the following form:

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

where each parenthetical list contains the names of variables and array elements that 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 that 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)
```

When you use array elements in EQUIVALENCE statements, they 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.

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

1. You cannot set two quantities declared in a COMMON statement to be equivalent to one another.

2. 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)
```

3. You cannot use EQUIVALENCE statements that cause the start of a common block to be extended backwards. 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. Write the EXTERNAL statement 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 FORTRAN defined functions.

You may also use FORTRAN defined function names for your 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 FORTRAN defined function, then the prefix is ignored.)

Note that specifying a predefined FORTRAN 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 properties 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 you 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:

1. Place parameter names only within the statement field of an initial or continuation line type, i.e., not within a comment line or literal text.
2. Place parameter names only where constants are acceptable.
3. Place parameter name references after the PARAMETER statement definition.
4. Use parameter names that are unique with respect to all other names in the program unit.
5. Do not redefine parameter names in subsequent PARAMETER statements.
6. Do not use parameter names 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) Write DATA statements as follows:

```
DATA List1/Data1/,List2/Data2/,...,Listn/Datan/
```

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 to the value 5, variable IB to the value 10, and the variable IC to the value 15. The number of storage locations you specify in the list of variables must be less than or equal to the number of storage locations you specify 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, array names, array elements, or labeled common variables. You may specify an entire array (unsubscripted array name) or a portion of an array in a DATA statement List as an implied DO loop construct. (See Section 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 you use an implied DO loop 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.

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

---

1. Refer to Section 6.5 for a description of labeled common.

You must identify octal constants by preceding them with a double quote (") symbol, e.g, "777.

You may specify literal data 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 you assign the same value to more than one item in List, a repeat specification may be used. Write the repeat specification 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, and L.

When the specified data type is not the same as that of the variable to which it is assigned, FORTRAN-20 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 to 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 that 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 'KLbbb'
```

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 'ƀƀƀƀƀ'. The string 'FGHIJ' is ignored.

CHAPTER 8

ASSIGNMENT STATEMENTS

8.1 INTRODUCTION

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

1. Arithmetic assignment statements

2. Logical assignment statements

3. Statement Label assignment (ASSIGN) statements.

8.2 ARITHMETIC ASSIGNMENT STATEMENT

You use statements of this type to assign specific numeric values to variables and/or array elements. Write arithmetic assignment statements in the form

```
v=e
```

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

In assignment statements, the equal 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-20 converts the value to the type of its assigned variable or array element. Table 8-1 describes the type conversion operations performed by FORTRAN-20 for each possible combination of variable and value types.

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	prohibited	R
DOUBLE-PRECISION	H	C,H,L	prohibited	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

Table 8-1 (Cont.)
Rules for Conversion in Mixed Mode Assignments

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 with 13 to 24 digits are termed double octal. Double octals require two storage locations. They are stored right-justified and are padded with zeros 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

Use this type of assignment statement to assign values to variables and array elements of type logical. Write the logical assignment statement 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 is replaced by logical truth.
F=.NOT.G	The contents of L is replaced by the logical complement of the contents of G.
M=A.GT.T or M=A>T	If A is greater than T, the contents of M is replaced by logical truth; if A is less than or equal to T, the contents of M is replaced by logical false. This can also be read: If A is greater than T, then M is true, otherwise, M is false.
L=((I.GT.H).AND.(J<=K))	The contents of L are replaced by either the true or false resultant of the expression.

8.4 ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT

Use the ASSIGN statement to assign a statement label constant, i.e., a 1- to 5-digit statement number, to a variable name. Write the ASSIGN statement in the 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, you may use any type of variable in an ASSIGN statement.

Once a variable has been assigned a statement number, FORTRAN-20 will consider it a label variable. If a label variable is used in an arithmetic statement, the result will be unpredictable.

ASSIGNMENT STATEMENTS

Use the ASSIGN statement in conjunction with assigned GO TO control statements (Chapter 9). The ASSIGN verb sets up statement label variables that 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
```

CHAPTER 9

CONTROL STATEMENTS

9.1 INTRODUCTION

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

1. GO TO
2. IF
3. DO
4. CONTINUE
5. STOP
6. PAUSE

9.2 GO TO CONTROL STATEMENTS

There are three kinds of GO TO statements:

1. Unconditional
2. Computed
3. Assigned

A GO TO control statement causes the statement that it identifies to be executed next, regardless of its position within the program. The following paragraphs describe each type of GO TO statement.

9.2.1 Unconditional GO TO Statements

Write GO TO statements of this type 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 transfers control of the program to the statement that it specifies.

You may position an unconditional GO TO statement anywhere in the source program except as the terminating statement of a DO loop.

9.2.2 Computed GO TO Statements

Write GO TO statements of this type 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. You may include any number of statement numbers in the list of this type of GO TO statement; however, each number you give 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 list. If the value of the expression 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 that 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

Write GO TO statements of this type 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 you give must be within the program or subprogram containing the GO TO statement.

Assigned GO TO statements of either foregoing form 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 list, 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

Write IF statements of this type in the form

```
IF(E)L1,L2,L3
```

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

This type of IF statement transfers control of the program to one of the given statements according to the computed value of the given expression. If the value of the expression is:

1. Less than 0, control is transferred to the statement identified by L1;
2. Equal to 0, control is transferred to the statement identified by L2;
3. Greater than 0, control is transferred to the statement identified by L3.

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

Examples

Sample Statement	
IF(ETA)4, 7, 12	Transfers 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	Transfers 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.

NOTE

You must label the statement following an arithmetic IF; otherwise the statement can never be executed.

9.3.2 Logical IF Statements

Write IF statements of this type in the form

```
IF(E)S
```

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

Logical IF statements transfer control of the program either to the next sequential executable statement or to 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 you give in a logical IF statement may be any executable statement except a DO statement or another logical IF statement.

Examples

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

9.3.3 Logical Two-Branch IF Statements

Write IF statements of this type in the form

```
IF (E) N1, N2
```

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

Logical two-branch IF statements transfer control of the program 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.

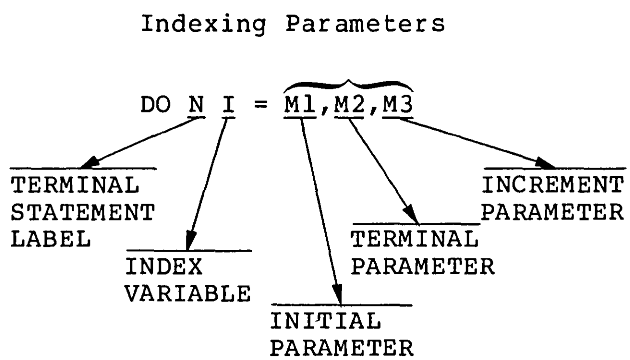
Note that you must number the statement immediately following the logical two-branch IF so that control can later be transferred to the portion of code that was skipped.

Examples

Sample Statement	
IF (LOG1) 10,20	Transfers control to statement 10 if LOG1 is negative; otherwise transfers control to statement 20.
IF (A.LT.B.AND.A.LT.C) 31,32	Transfers control to statement 31 if A is less than both B and C; transfers 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; write them in the following form:



where

1. 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 that follows the DO statement up to and including its specified terminal statement.

2. Index Variable I is an unsubscripted variable whose value 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 available for use in the program when:

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

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

   c. if an input-output statement with either or both the options END= or ERR= (Chapter 10) appears within the DO statement range.

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

4. Terminal Parameter M2 provides the value that determines how many repetitions of the DO statement range are performed.

5. Increment Parameter M3 specifies the value to be added to the initial parameter (M1) on completion of each cycle of the DO loop. If M3 and its preceding comma are omitted, M3 is assumed to be equal to 1.

An indexing parameter may be any arithmetic expression resulting 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 execute is specified by the formula:

```
MAX(1,((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 by 1. If the number of specified iterations has not been performed (i.e., if the count variable is still negative), 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 passes 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 (Section 9.4.3).

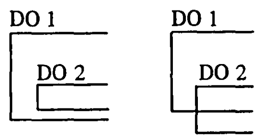
9.4.1 **Nested DO Statements**

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.

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

   Example

   Valid Invalid

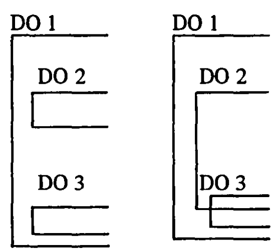


The range of DO 2 is outside that of DO 1.

2. The ranges of nested DO statements cannot overlap.

   Example

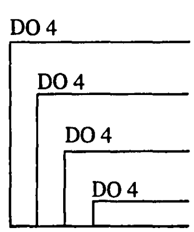
   Valid Invalid



The ranges of loop DO 2 and DO 3 overlap.

3. 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 other than within the range of the DO statement to which it belongs.

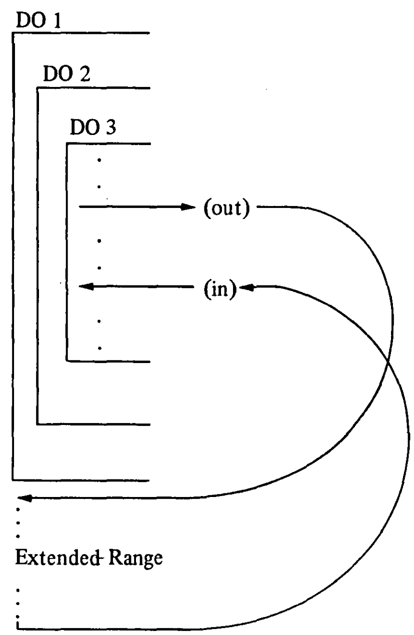
   Example



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

9.4.2 Extended Range

The extended range of a DO statement is defined as the set of statements that execute between the transfers out of the innermost DO statement of a set of nested DOs and the transfer back into the range of this innermost DO statement. The extended range of a nested DO statement is as follows:



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

1. 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.

2. 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.

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

4. The extended range of a DO statement cannot change the index variable or inde.xing parameters of the DO statement.

5. You may use and return from a subprogram within an extended range.

9.4.3 Permitted Transfer Operations

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

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

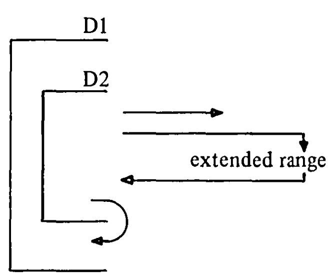
2. A transfer into the range of a DO statement is permitted if it is made from the extended range of the DO statement.

3. You may use and return from a subprogram from within the range of any:

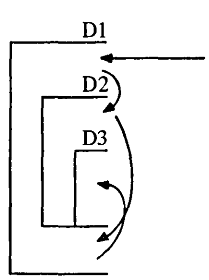
   a. DO loop,
   b. nested DO loop, or
   c. extended range loop (in which you leave the loop via a GO TO, execute statements elsewhere, and return to the original loop).

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

Valid Transfers



Invalid Transfer



9.5 CONTINUE STATEMENT

You may place CONTINUE statements 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. Write this statement 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, HEADING,PAGENO
       .
       .
       .
```

9.6 STOP STATEMENT

Execution of the STOP statement causes the execution of the object program to be terminated and returns control to the monitor. A descriptive message may optionally be included in the STOP statement to be output to your I/O terminal immediately before program execution is terminated. Write this statement like this:

```
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 your I/O terminal when the STOP statement executes. The string N 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

Execution of a PAUSE statement suspends the execution of the object program and gives you the option to:

1. Continue execution of the program
2. Exit
3. Initiate a TRACE operation (Section 9.7.1).

The permitted forms of the PAUSE statements are:

1. PAUSE
2. PAUSE 'literal string'
3. PAUSE n, where n is an octal string up to 12 digits.

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 your terminal. If the form of the PAUSE statement contains either a literal string or an integer constant, the string or constant prints on a line preceding the standard message. For example, the statement

```
PAUSE 'TEST POINT A'
```

causes the following to be printed at your 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 your 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 a complete history of all subroutine calls made during the execution of the program, up to the execution of the PAUSE statement to be printed at your terminal. The history printed by the TRACE routine consists of:

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

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

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

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

5. An alphabetic code (written within square brackets) that specifies the types 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.  (414056)  <<---  PAUS.+141(376)   <#1>        [U]
PAUS.   (235)     <<---  MAIN.+4(151)     <#1>        [U]
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE.
*G
```

In addition to its use with the PAUSE statement, you may call the TRACE routine 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 prints 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 as described in the preceding paragraph.

CHAPTER 10

I/O STATEMENTS

10.1 DATA TRANSFER OPERATIONS

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

1. A list in the I/O statement that initiated the transfer

2. A list defined by a NAMELIST statement, or

3. 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 (list-directed I/O).

The following sections describe the statements and data format required to initiate I/O transfer operations.

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 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 access to and transfer of records 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.

You may use the FORTRAN-20 OPEN statement - see Chapter 12 - or a subroutine call to DEFINE FILE to set up random access files.

Use the OPEN statement 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: Use it only for sequential output (write) operations. The append mode permits you 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.

You must use an OPEN statement 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 (READ or WRITE)
u	= logical unit number
f	= FORMAT statement number in the current program unit 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

The following paragraphs provide details of the foregoing components.

10.3.1 I/O Statement Keywords

The keywords (names) of the FORTRAN-10 I/O statements described in this chapter are:

1. READ	6. WRITE
2. REREAD	7. PRINT
3. ACCEPT	8. TYPE
4. FIND	9. ENCODE
5. DECODE	

10.3.2 FORTRAN Logical Unit Numbers

Decimal numbers identify the physical devices used for most FORTRAN I/O operations. 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 you cannot access.

You may make the logical device assignments at run time, or you may use the standard assignments contained by the FORTRAN-20 Object Time System (FOROTS). Table 10-1 lists the standard logical device assignments. We recommend that you specify the device explicitly in the OPEN statement.

10.3.3 FORMAT Statement References

A FORMAT statement contains a set of format specifications that defines the structure of a record and the form of the data fields comprising 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 that contains format specifications causes the structure and data of the transferred record to assume the form specified in the referenced format. 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 FORTRAN-20 embedded control words; the control words are invisible to you. You cannot prepare files of this type without using FOROTS. Unformatted files are for use only within the FORTRAN environment.

Table 10-1
FORTRAN-20 Logical Device Assignments

Device/Function	Default Filename	FORTRAN Logical Unit Number	Use
Standard Devices*			
0		00	ILLEGAL
DSK	FORxx.DAT	01	Disk
CDR		02	Card Reader
LPT		03	Line Printer
CTY		04	Console Teletype
TTY		05	User's Teletype
		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	

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

Table 10-1 (Cont.)
FORTRAN-20 Logical Device Assignments

Device/Function	Default Filename	FORTRAN Logical Unit Number	Use
Standard Devices*			
DEV1	FORxx.DAT	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
		-2	Not Valid
LPT	FORLPT.DAT	-3	PRINT statement
TTY	FORTTY.DAT	-1	TYPE statement

*The total number of standard devices permitted is an 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 (Section 10.3.4.1), which specify sets of array elements, may also be included in I/O lists. The number of items in a statement 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 (Section 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.

Write implied DOs 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 or an array element identifier
I	= the index control variable that may represent 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

S must be an integer variable

(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.
(I,I=1,5)	Specifies the integers 1,2,3,4, and 5.

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

Example (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.

You may also nest implied DO constructs. Nested implied DOs 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 you specify an array or set of array elements 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 dimensioned 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)
```

10.3.4.2 **Formatted Record Handling** - Data is processed under format control so that each item in the I/O list is matched with a field descriptor in the FORMAT statement. If the end of the FORMAT specification is reached and more items remain in the I/O list, a new line or record is established and the data processing is restarted, either:

1. at the first item in the FORMAT specification or,

2. (if parenthesized sets of FORMAT specifications exist within the FORMAT specification) with the last set within the FORMAT specification.

On input, if the record is exhausted before the data transfers are completed, the remainder of the transfer is completed as if the record were extended with blanks. See Section 13.2.2 for more details.

10.3.5 Specification of Records for Random Access

You must identify records to be transferred in a random access mode in an I/O statement by an integer expression or variable preceded by an apostrophe used as a delimiter, e.g., '101.

NOTE

> You may use a pound sign (#) in place of the apostrophe ('), e.g., both #101 and '101 are accepted by FORTRAN-20.

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 the 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
```

You may use list-directed transfers to read data from any acceptable input device, including an input keyboard terminal.

NOTE

Do not use device positioning commands, such as BACKSPACE, SKIP RECORD, etc., in conjunction with list-directed I/O operations. If you do, the results are unpredictable.

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

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

2. Literal constants must be enclosed within single quotes, e.g., 'ABLE'. A quoted string which is too long to fit in one element of the input list will be placed in adjacent elements and will be padded with blanks. If a quoted string is being placed in an array and it fills more than one element of the array, the remaining elements of the array will be unchanged. In this case, it is assumed that the user meant for the long string to go into the array and for any following data to go into the rest of the input list. If the string fits into one element of the array, the array will continue to be filled.

3. Blanks are delimiters; therefore, they are not permitted in any but literal constants.

4. You may omit decimal points from real constants that do not have a fractional part. FORTRAN-20 assumes that the decimal point follows the rightmost digit of a real constant.

5. Complex constants must be enclosed in parentheses.

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

1. Delimiters may be either commas or blanks.

2. 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-20 as being only a single delimiter.

3. Represent a null (the complete absence of a datum) by two consecutive commas that have no intervening constant(s). You may place any number of blanks, tabs, carriage return/line feed characters, or end-of-input conditions between the commas of a null. Each time you specify a null item in the input data, its corresponding list element is skipped (unchanged). The following illustrates the effect of a null input:

```
INPUT Items              101,'A',tab,'NO1',
                          /   /   /    /
                         v   v   v    v
Corresponding            A  ,LIT,IAB,NUMBER
I/O List Items           \   |   /    /
                          v  v  v    v
Resulting                101.A un-  NO1
Contents of                   changed
List Items                      IAB
```

4. Slashes (/) cause the current input operation to terminate even if all the items of the directing list are not filled. The contents of items of the directing I/O list that 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.

5. Once the I/O list has been satisfied (values have been transferred 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 so that an item is 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

You may define one or more lists 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. By using the NAMELIST statement, you eliminate the need for specifying the entire I/O list.

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

You may use only NAMELIST-controlled READ/WRITE statements to bring in/write out records formatted in the following manner. Format records for NAMELIST-controlled input operations as follows:

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

where

1. $ 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.

2. 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.

3. D1 through Dn are pairs of the form "variable=value" where the value is assigned to the associated variable. These items cannot be octal numbers or literal strings.

NOTE

Do not use device positioning commands such as BACKSPACE, SKIP RECORD, etc., in conjunction with NAMELIST-controlled I/O operations. If you do, the results are unpredictable.

See Chapter 11 for more information on NAMELIST I/O transfers.

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

You may optionally add either or both an error exit or an end-of-file argument to the portion in parentheses of any form of the READ and WRITE statements when a unit is specified.

Write the error exit argument as ERR=c where c is a statement number in the current program unit. Using this argument terminates the current I/O operation and transfers program control to the statement identified by the argument if an 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. See the Library Subroutine ERRSNS (Chapter 15) to find out how to identify the actual error that occurred.

When an ERR= transfer occurs, all items on the input list and all implied DO indexes on input or output lists become undefined.

Write the end-of-file argument as END=d, where d is a statement number in the current program unit. This branch, when taken, stops the current I/O operation and transfers program control to the statement identified by the argument. In the example below, the detection of an end-of-file condition during the execution of

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

results in the transfer of control to statement 50.

When an END= transfer occurs, all items on the input list receive the value zero and all implied DO indices on input lists become undefined.

If the END= argument is not present, but an ERR= argument is, an end-of-file (EOF) condition is treated as a user-trappable error. If neither the ERR= nor the END= argument is present and an end-of-file condition is detected, a message is printed, the file is closed, program execution is terminated, and control is returned to the monitor.

10.5 READ STATEMENTS

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

10.6 SEQUENTIAL FORMATTED READ TRANSFERS

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

1. Form: READ (u,f)list

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

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

2. 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)

3. 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

4. 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 as specified in f.

   Example: READ 15, ARRAY (20,30)

10.6.1 Sequential Unformatted Binary READ Transfer

You may use only the following form of the READ statement for the sequential transfer of unformatted input of 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. You may read only binary files output by a FORTRAN-20 unformatted WRITE statement with this type of READ statement.

NOTE

If you use the form READ (u), one unformatted input record will be skipped.

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

10.6.2 Sequential List-Directed READ Transfer

You may use the following forms of the READ statements to control a sequential, list-directed input transfer:

1. Form: READ(u,*)list

   Use: Read data from logical device u into processor storage as 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

2. Form: READ *,list

   Use: Read the data from the READ default device (card reader) 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

10.6.3 Sequential NAMELIST-Controlled READ Transfers

You may use only the following form of the READ statement to initiate a sequential NAMELIST-controlled input transfer:

Form: READ(u,N)

Use: Read data from logical unit u into processor storage as the value of the locations identified by the NAMELIST input 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.

10.6.4 Random Access Formatted READ Transfers

You may use only the following form of the READ statement 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, an error results.)

Example: READ(1#20,100) I, X(J)

10.6.5 Random Access Unformatted READ Transfers

You may use only the following form of the READ statement 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, an 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 logical input record R to be skipped.

10.7 SUMMARY OF READ STATEMENTS

Table 10-2 summarizes the various forms of the READ statements.

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: You may include the ERR=c and END=d arguments 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).		

10.8 REREAD STATEMENT

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

You cannot use the REREAD feature of FORTRAN-20 until an input (READ) transfer from a file has been accomplished. If you use REREAD prematurely, an error results.

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, you may use the same or new format specification by each successive REREAD statement.

You may use the REREAD statement only for sequential formatted data transfers. 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),
      1    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 statement 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.

NOTE

If you try to REREAD a record input from the teletype, you will get either the current record or the last 150 characters of the current record, whichever is the lesser.

10.9 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.

10.9.1 Sequential Formatted WRITE Transfers

You may use the following forms of the WRITE statement for the sequential transfer of formatted data:

1. 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 f.

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

2. Form: WRITE f,list

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

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

3. Form: WRITE f

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

   Example: WRITE 10

10.9.2 Sequential Unformatted Binary WRITE Transfer

You may use the following form of the WRITE statements 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)

10.9.3 Sequential List-Directed WRITE Transfers

You may use the following form of the WRITE statement 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 taken.

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

10.9.4 Sequential NAMELIST-Controlled WRITE Transfers

You may use only the following form of the WRITE statement 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 into the file associated with logical unit u.

Example: WRITE(12,NMLST)

10.9.5 Random Access Formatted WRITE Transfers

You may use only the following form of the WRITE statement 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 the file associated with logical device u. Only disk files that have been set up by either an OPEN statement or a call to the subroutine DEFINE FILE may be accessed by a WRITE transfer of this form. The data transferred will be formatted according to the specifications given in f. Only those records that have been specifically written are available to be read.

10.9.6 Random Access Unformatted WRITE Transfers

You may use only the following form of the WRITE statement 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 file associated with logical device unit u. Only disk files that 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 that have been specifically written are available to be read.

10.10 SUMMARY OF WRITE STATEMENTS

Table 10-3 summarizes the various forms of the WRITE statements.

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: You may include the ERR=c and END=d arguments in any WRITE statement which has a unit number; however, they must be last.		

10.11 ACCEPT STATEMENT

The ACCEPT statement enables you to input data via either a terminal keyboard or a batch control file directly into specified processor storage locations. Use this statement only in the sequential transfer mode for the formatted transfer of inputs from your terminal keyboard during program execution. The following paragraphs describe the permitted forms of the ACCEPT statement.

10.11.1 Formatted ACCEPT Transfers

Use the following forms of the ACCEPT statement for the sequential transfer of formatted data.

1. Form: ACCEPT f,list

   Use: Input data character-by-character from the user's terminal 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)

2. Form: ACCEPT *,list

   Use: Input data character-by-character from the user's terminal 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

10.11.2 ACCEPT Transfers Into FORMAT Statements

You may use the following form of the ACCEPT statement to input data from your terminal keyboard directly into a specified FORMAT statement if the FORMAT statement has either or both a Hollerith (H), or a literal ('s') field descriptor. If the referenced statement has neither of the foregoing 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

10.12 PRINT STATEMENT

The PRINT statement causes data from specified processor storage locations to be output on the standard output device (line printer). Use this statement only for sequential formatted data transfer operation; write it in either of the three following forms:

1. 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 f.

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

2. 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 taken.

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

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

Example

The sequence

```
55    FORMAT(' END OF ROUTINE')
      .
      .
      .
      PRINT 55
```

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

10.13 TYPE STATEMENT

The TYPE statement causes data from specified processor storage locations to be output to your (control) terminal printing or display device. Use this statement only for sequential formatted data transfers; write it in one of the following forms:

1. Form: TYPE f,list

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

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

2. Form: TYPE f

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

   Example: TYPE 101

3. Form: TYPE *,list

   Use: Output the values of the processor storage locations identified by the contents of list to 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 taken.

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

10.14 FIND STATEMENT

The FIND statement does not initiate a data transfer operation; use it during random access read operations to locate the next record to be read while the current record is being input. The 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 processed until the second READ statement in the sequence is executed.

10.15 ENCODE AND DECODE STATEMENTS

Use the ENCODE and DECODE statements 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 storage area. ENCODE operations are similar to those performed by a WRITE statement.

The DECODE statement transfers data from a specified 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.

Write the ENCODE and DECODE statements 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

5 characters per storage location are stored in the buffer 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 starts on a new storage location boundary rather than there being a CRLF inserted between records.

10.15.1 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 given in f. The converted characters are then written into the destination area starting at location s. If you try to transfer more characters than the specified area can contain, the excess characters are ignored.

If you transfer fewer characters than specified for the record size, the empty character locations are filled with blanks.

Example: ENCODE(500,101,START)TABLE

10.15.2 DECODE Statement

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

Form: DECODE(c,f,s)list

Use: The character strings are taken starting at location s, converted (decoded) according to the format specifications given in f, and stored as the values of the processor storage 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)

10.15.3 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 number 300.45
A(2) contains the floating point number 3.0
J    is an integer variable
B    is a 4-word array of indeterminate contents
C    contains the ASCII string 12345
```

```
(1)        DO 2 J=1,2
(2)        ENCODE(16,10,B)J,A(J)
(3)   10   FORMAT(1X,2HA(,I1,4H) = ,F8,2)
(4)        TYPE 11,B
(5)   11   FORMAT(4A5)
(6)   2    CONTINUE
(7)        DECODE(5,12,C)B
(8)   12   FORMAT(3F1.0,1X,F1.0)
(9)        TYPE 13,B
(10)  13   FORMAT(4F5.2)
(11)       END
```

Array B can contain 20 ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

```
B(1) =        'A(1) '          Typed at line 4 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 at line 4 as
B(2) =        '=    '
B(3) =        '3.0  '          A(2) =     3.0
B(4) =        '     '
```

The DECODE statement:

1. Extracts the digits 1, 2, and 3 from C
2. Converts them to floating point values
3. Stores them in B(1), B(2), and B(3)
4. Skips the next character (the digit 4)
5. Extracts the digit 5 from C
6. Converts it to a floating-point value, and,
7. Stores it in B(4)

The output from the TYPE statement at line 9 is:

```
    1.00 2.00 3.00 5.00
```

10.16 SUMMARY OF I/O STATEMENTS

Table 10-4 summarizes all permitted forms of the I/O statement.

Table 10-4
Summary of I/O Statements

I/O Statements	Formatted	Transfer Format Control		List-Directed
		Unformatted	Namelist	
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 ACCEPT f			ACCEPT *,list

1. You must use an OPEN statement to set up an append mode.

2. You must use either the OPEN statement or a call to the DEFINE FILE subroutine to set up a random access mode.

Table 10-4 (Cont.)
Summary of I/O Statements

I/O Statements	Formatted	Transfer Format Control		List-Direction
		Unformatted	Namelist	
PRINT Sequential only	PRINT f,list PRINT f			PRINT *,list
TYPE Sequential only	TYPE f,list 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
- f — statement number of FORMAT statement or name of array containing format information
- list — I/O list
- N — name of specific NAMELIST I/O list
- * — symbol used to specify list-directed I/O operator
- #R — variable which specifies logical record position
- c — number of characters per internal record
- s — address of the first storage location to be used

CHAPTER 11

NAMELIST STATEMENTS

11.1 INTRODUCTION

Use the NAMELIST statement to define I/O lists similar to those described in Chapter 10 (Paragraph 10.3.4). Reference defined NAMELIST I/O lists 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

Write NAMELIST statements in the following form:

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

where

/N1/ through /Nn/	represents names of individual lists. Always enclose the name with 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. Delimit the items of a list by commas. Each list of a NAMELIST statement is identified (and referenced) by the name immediately preceding the list.

Example

```
DIMENSION C(2,4)
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 consisting of the array TOTAL.

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

The rules for structuring a NAMELIST statement are:

1. You may use a maximum of six characters for a NAMELIST name.

2. You must begin it with an alphabetic character.

3. You must enclose it in slashes.

4. The NAMELIST name must precede the list of entries to which it refers.

5. The NAMELIST name must be unique within the program.

6. You may define a NAMELIST name only once, and you must define it by a NAMELIST statement. Once defined, you may use a name only in READ or WRITE statements.

7. You must define the NAMELIST name in advance of the I/O statement in which it is used.

8. You cannot use a variable used in a NAMELIST statement as a dummy argument in a SUBROUTINE definition.

9. You must define any dimensioned variable contained in a NAMELIST statement in an array declaration statement preceding the NAMELIST statement.

11.2.1 NAMELIST-Controlled Input Transfers

During input (READ) transfer operations in which a NAMELIST-defined name is referenced, the records are read until a record is found that begins with the sequence ' $' (a space followed by a dollar sign) followed by the referenced name. The dollar sign must be the second character in the record; the first character in the record must be a blank. 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 variable 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

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

2. 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 (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.

NOTE

"&" may be used instead of "$" in NAMELIST-controlled input.

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:

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

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

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

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

generates the following form of output:

```
Column
|
|
v
 $NAME1
  JOE=  -6.750000     ,    0.2340000E-04,     680.0000     ,    -17.80000
   0.0000000E+00,   -1970000.     ,  K1=   73.10000     ,
  ALPHA=   3.000000     , $
```

NOTE

Do not use device positioning commands such as BACKSPACE, SKIP, RECORD, etc., with NAMELIST-controlled I/O operations. If you do, the results are unpredictable.

CHAPTER 12

FILE CONTROL STATEMENTS

12.1 INTRODUCTION

This chapter describes the OPEN and CLOSE statements.

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

12.2 OPEN AND CLOSE STATEMENTS

Both the OPEN and CLOSE statements are unique to FORTRAN-20; they both use the same format and have the same options and arguments.

The OPEN statement enables you to define all of the important aspects of each desired data transfer operation; it provides an extensive list of required and optional arguments that define in detail:

1. the name and location of the data file
2. the type of access required
3. the data format within the file
4. the protection code(1) to be assigned an output data file
5. the disposition of the data file
6. data file record, block and file sizes
7. a data file version identifier

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

The general form of the OPEN statement is:

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

---

1. Refer to the Monitor Calls Manual, for a description of file access protection codes.

Use the CLOSE statement 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. Note that the CLOSE statement can change the name, protection, directory, and disposition of the file being closed.

Once a CLOSE statement has been executed, you must use another OPEN statement to regain access to the closed file.

The general form of the CLOSE statement is:

```
CLOSE(Argl.,Arg2.,...,Argn)
```

CAUTION

If you use a filename argument in a CLOSE statement that is different from the current filename, the file will be renamed.

12.2.1 Options for OPEN and CLOSE Statements

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

1. 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

   NOTE

   FORTRAN-20 standard logical unit assignments are described in Chapter 10 (Table 10-1). The range, i.e., 1-63, of the possible UNIT numbers is an installation-defined parameter.

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

3. ACCESS

ACCESS describes the type of input and/or output statements and the file access mode to be used in a specified data transfer operation. You may assign ACCESS 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:

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

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

c. SEQINOUT — The specified data file may be first read, then written (READ/WRITE sequence) record-by-record in a sequential access mode. When you specify SEQINOUT, a WRITE/READ sequence is illegal. If no access is specified, SEQINOUT is assumed.

d. 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. The RECORD SIZE option is required for random access operations. You must specify a disk device when the random argument is used.

e. RANDIN — This argument enables you to establish a special, read-only random access mode with a named file. During a RANDIN mode, you 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.

f. APPEND — The record specified by a corresponding WRITE statement is to be added to the logical end of a named file. You must close and then reopen the modified file to permit it to be read.

The general form of the ACCESS argument is:

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

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

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

NOTE

Refer to the Monitor Calls Manual for a detailed description of the data modes given in the following list.

You must use one of the following character set specifications 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.
'DUMP'	The data file to be transferred is to be handled in a DUMP mode of operation.

The general form of the MODE argument is:

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

5. DISPOSE

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

Literal	Action Indicated
'SAVE'	Leave the file on the device.
'DELETE'	If the device involved is a 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:

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

6. FILE

This option specifies the name of the file involved in the data transfer operation. FILE must be either a 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 extension. Any excess characters in either the name or extension are ignored. If you omit the period and extension, the extension .DAT is assumed; if just the extension is omitted, a null extension is assumed. So if you want a filename without an extension, remember to use the period.

If a filename is not specified or is zero, a default name is generated that 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= A literal or variable set to a literal

7. 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 (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 installation is used. The general form of the PROTECTION argument is:

PROTECTION= 3-digit octal constant or integer variable

8. DIRECTORY

Use this option for disk files only. It specifies the location of the user file directory (UFD) or the sub-file directory (SFD) that contains the file specified in the OPEN statement. A directory identifier may consist of either:

a. Your project programmer number that identifies the UFD, for example, 10,7, or

b. A UFD/SFD directory path specification. A path specification lists the UFD and the names of its SFDs that form a path to the desired SFD. For example, the following path specification identifies the path leading to SFD 1234:

```
10,7,SFDA,SFDB,1234
```

NOTE

Refer to the Monitor Calls Manual for a complete description of directories and multilevel directory structures.

The general form of a DIRECTORY argument is:

DIRECTORY= Literal or array name containing directory path specification

You may also establish an array containing the directory specification as its elements and reference the array in the DIRECTORY argument. Single-precision arrays permit 5-character directory names to be used; double-precision arrays permit 6-character names to be used. You must use a zero (0) entry to terminate a directory path specification given in an array.

Examples of the use of single- and double-precision arrays in an OPEN statement DIRECTORY specification follow:

a. Single-Precision Array

```
OPEN (UNIT = 5, DIRECTORY = PATH,...)
```

where PATH and its elements are:

```
DIMENSION PATH (5)
PATH (1)= "10 !(PROJECT NUMBER)
PATH (2)= "7  !(PROGRAMMER NUMBER) UFD
PATH (3)='SFDA'  Names of sub-file
PATH (4)='SFDB'  directories (SFD's)
PATH (5)=0
```

b. Double-Precision Array

```
OPEN (UNIT=5, DIRECTORY = PATH,...)
```

where PATH and its elements are:

```
DOUBLE PRECISION PATH (5)

PATH (1)="000000000010000000000007
            !(PROJ.,PROG. NUMBERS=UFD)

PATH (2)='SFDABC'
PATH (3)='MYAREA'   !names of sub-file
PATH (4)='YOURIT'   !directories (SFDs)
PATH (5)=0
```

The elements of a directory specification may then be either a literal or a single- or double-precision array.

The following is an example of a literal specification:

```
DIRECTORY='10,7,SFD1,SFD2,SFD3'

  Project      Sub-File
  Programmer   Directory
  Number       Path
```

Whenever the specification is an array, you may specify the required project and programmer numbers either of two ways. You can use one word with the project number in the left half and the programmer number in the right half, or, use the right halves of separate sequential word locations.

9.	BUFFER COUNT	This option enables you 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
10.	FILE SIZE	Use this option for disk operations only; it enables you to estimate the number of words that an output file is going to contain. The use of FILE SIZE enables you 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 and will be rounded up to the next higher block boundary (multiple of 200 octal). The general form of this argument is: FILE SIZE= An integer constant or variable
11.	VERSION	Use this option for disk operations only; it enables you 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
12.	BLOCK SIZE	You can use this option for all storage media except disk. It enables you to specify a physical storage block size for a device. 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
13.	RECORD SIZE	This option enables you 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 number 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

14. ASSOCIATE VARIABLE

Use this option 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 sequentially access records starting from the current READ. For example, if record number 3 were read, the ASSOCIATE VARIABLE would be equal to 4. The general form of this argument is:

ASSOCIATE VARIABLE = Integer variable

15. PARITY

Use this option for magnetic tape operations only; it permits you 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)
```

16. DENSITY

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

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

17. DIALOG

The use of this option in an OPEN statement enables you 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 your terminal when the OPEN statement is executed. FOROTS outputs the following messages at the user's terminal.

```
UNIT=n:/ACCESS=SEQINOUT/MODE=ASCII
ENTER NEW FILE SPECS. END WITH A $ (ALT)
```

Once the message and defined file specification are output, you may enter any desired changes. You need enter only the arguments that are to be changed.

The second form of the argument is:

DIALOG= Literal or array

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

18. ERR

The use of this option in an OPEN or CLOSE statement enables you to transfer program control to an executable statement when an error is detected during the processing of the OPEN or CLOSE statement. The general form of this option is:

ERR= s

where s is the statement label of an executable statement (that appears in the same program unit as the error specifier) to which program control is transferred when an error is detected.

Associated with the ERR= option on OPEN/CLOSE is the subroutine ERRSNS that enables you to pinpoint the error. See Appendix H for FOROTS error values returned by ERRSNS.

Examples:

```
OPEN (UNIT= 1, DEVICE= 'DSK', ACCESS= 'SEQIN', MODE= 'BINARY')
```

causes a disk file named FOR01.DAT (since no FILE= option was specified) to be opened on unit 1 for sequential input in binary mode.

```
OPEN (UNIT= 3, DEVICE= 'DSK', FILE= 'PAYROL.DAT',
1     ACCESS= 'RANDOM', MODE= 'ASCII', RECORD SIZE= 80,
2     ASSOCIATE VARIABLE= I, ERR= 240)
```

causes a disk file named PAYROL.DAT to be opened on unit 3 for random input/output operations in ASCII mode. The records in PAYROL.DAT are 80 characters long; the ASSOCIATE VARIABLE for this file is I. If an error occurs during the execution of this OPEN statement, the OPEN will terminate and control will transfer to the statement labeled 240.

```
CLOSE (UNIT= 3, DISPOSE= 'DELETE')
```

causes the file on unit 3 to be closed and removed if the file is on disk.

12.2.2 Summary of OPEN/CLOSE Statement Options

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

Table 12-1
OPEN/CLOSE Statement Arguments

Argument	Possible Value	Open*	Close*
ACCESS=	'SEQIN', 'SEQOUT', 'SEQINOUT', 'RANDIN', 'RANDOM', 'APPEND', or variable	O	I
ASSOCIATE VARIABLE=	Integer variable	O	I
BLOCK SIZE=	Integer constant or variable	O	I
BUFFER COUNT=	Integer constant or variable	O	I
DENSITY=	Literal constant or variable	O	I
DEVICE=	Literal constant or variable	O	I
DIALOG=	Literal or array or none	O	I
DIRECTORY=	Literal or variable or array	O	O
DISPOSE=	Literal constant or variable	O	O
ERR=	Statement Number	O	O
FILE=	Literal constant or variable	O	O
FILE SIZE=	Integer constant or variable	O	I
MODE=	Literal constant or variable	O	I
PARITY=	Literal constant or variable	O	I
PROTECTION=	An octal constant or integer variable	O	O
RECORD SIZE=	Integer constant or integer variable	O	I
UNIT=	Integer variable or constant	R	R
VERSION=	Octal constant or variable	O	O

*
R = Required
O = Optional
I = Ignored

CHAPTER 13

FORMAT STATEMENT

13.1 INTRODUCTION

Use FORMAT statements in conjunction with the I/O list of I/O statements during formatted data transfer operations. The FORMAT statements contain field descriptors that, together with the list items of associated I/O statements, specify the forms of the data and data fields that comprise each record.

FORMAT statements may appear almost anywhere in a 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.)

You must label FORMAT statements so that I/O statements can reference them.

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 containing 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.

You may represent repeated field descriptors or groups of field descriptors by using a repeat form. Indicate the repetition of a single field descriptor by preceding the descriptor with an integer constant that 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))
```

You may nest the repeat forms of field descriptors 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 following paragraphs discuss the manner in which you may use the foregoing statement forms and the effect each has on the data involved.

13.2 FORMAT DESCRIPTORS

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

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

where

- r = an optional unsigned integer representing a repeat count. This option enables a field descriptor to be repeated r times.
- w = an optional integer constant representing 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 specifying the number of fractional digits that 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 specifying 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. Table 13-1 gives the FORTRAN-20 conversion codes and a brief description of the function of each.

Table 13-1
FORTRAN-20 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 you wish to specify

```
FORMAT(I2,2I5)
```

The following paragraphs provide detailed descriptions of the various types of format descriptors, the manner in which they are written and employed, and their use in FORMAT statements.

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 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 or double-precision 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
.
.
.
```

Table 13-2 describes the actions performed by several types of formatted WRITE statements on the data given in the foregoing program segment.

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.nn	18.10
4	Iw	I5	WRITE(-,-)I1	aaaan	b̸b̸506
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	b̸b̸b̸18.10
10	Gw	G5	WRITE(-,-)I1	aaan	b̸b̸506

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 (b̸) or a minus sign depending on the numeric output.

Notes:

1. In Item 1, the value D1 has only two 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 that 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.

Table 13-3 summarizes the internal and external forms of the data specified by the numeric format conversion code.

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, decimal integer, logical (T or F), or complex (two decimal floating-point numbers), depending upon the internal form

Complex quantities transfer 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 Variables

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 that comprise the referenced FORMAT statement's format specifications.

In processing each FORMAT controlled I/O statement that 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, Hollerith field descriptors, and spacing descriptors, but not past data field descriptors.)

A record is terminated by one of the following:

1. a slash in the FORMAT specification
2. the delimiting right parentheses,), of the FORMAT statement
3. a lack of items in the I/O list
4. a lack of Hollerith or literal 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, 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 a pair of consecutive slashes, //, or a single slash followed by the end of the FORMAT statement is encountered, an empty record is written.

13.2.3 G, General Numeric Conversion Code

You may use the G conversion code 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 type G 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 conversion performed for various ranges of magnitude (external form) of real and double-precision data.

13.2.4 Numeric Fields with Scale Factors

You may add scale factors 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 you add a scale factor to an type F field descriptor (or type G if the external field is a fixed point decimal) a power of 10 is specified so that

External Form of Number = (Internal Form)*10**(scale factor)

For example, assuming the data involved to be the real number 26.451, the field descriptor

F8.3

produces the external field

b̸b̸26.451

Table 13-4
Descriptor Conversion of Real and Double-Precision
Data According to Magnitude

Magnitude of Data in External Form (M)	Equivalent Method of Conversion Performed
0.1 M⩽1	F(w-4).d,4X
1 M⩽10	F(w-4).(d-1),4X
.	.
.	.
.	.
.	.
10d-2 M⩽10d-1	F(w-4).1,4X
10d-1 M⩽10d	F(w-4).0,4X
ALL OTHERS	Ew.d

NOTE

In all numeric field conversions, the field width (w) you specify 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

    b̸b̸b̸2.645

When you add a scale factor to D, E, and G (external field not a decimal fixed-point) type field descriptors, it multiplies the number by the specified power of ten and the exponent is changed accordingly.

In input operations, type F (and type G, if the external field is decimal fixed-point) conversions are the only ones affected by scale factors.

When you specify no scale factor, it is understood to be zero. Once you specify a scale factor, however, it holds for all subsequent types D, E, F, and G field descriptors within the same format specification unless another scale factor is specified. A scale factor is reset to zero when you specify a scale factor of zero. Scale factors have no effect on I and O type field descriptors.

When you add a scale factor to a D or E field descriptor, it specifies a power of 10 so that the external form of the number has its mantissa multiplied by the specified power of 10; its exponent is adjusted accordingly.

For example, assuming the data involved to be the real number 12.493, the field descriptor

    E11.3

produces the external field

    b̸b̸0.125E+02

The addition of the scale factor 2P

    2PE11.3

produces the external field

    b̸b̸12.49E+00

With a scale factor of zero, the number of significant digits printed by a format of the form

    Ew.d

or

    Dw.d

is the number of digits to the right of the decimal point.

For a negative scale factor nP, for $-d<n<0$, there will be ABS(n) leading zeros and d-ABS(n) significant digits after the decimal point, for a total of d digits after the decimal point. If $n \leq -d$, there will be d insignificant digits (zeros) to the right of the decimal point.

If the scale factor nP is positive, for $0<n<d+2$ there will be n significant digits to the left of the decimal point and d-n+1 significant digits to the right of the decimal point (for a total of

d+1 significant digits). If n⩾d+2, there will be d+1 significant digits and n-d-1 insignificant trailing zeros on the left of the decimal point.

If the data to be printed is 12.493, these formats produce results as follows:

FORMAT	OUTPUT	SIGNIFICANT DIGITS	REASON
E15.3	b̸b̸b̸b̸b̸b̸0.125E+02	3	n=0
1PE15.3	b̸b̸b̸b̸b̸b̸1.249E+01	4	n<d+2
-1PE15.3	b̸b̸b̸b̸b̸b̸.012E+03	3	-d<n
2PE15.3	b̸b̸b̸b̸b̸b̸12.49E+00	4	n<d+2
-3PE15.3	b̸b̸b̸b̸b̸b̸0.000E+05	0	n⩽-d
4PE15.3	b̸b̸b̸b̸b̸b̸1249.E-02	4	n<d+2
6PE15.3	b̸b̸b̸b̸124900.E-04	4	n⩾d+2

13.2.5 Logical Field Descriptors

You may transfer logical data 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, the w.d portion of the specification so that can be 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

---

1. If d is given, w must also be specified.

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,,TRUEXXX1ƀƀƀƀABC
```

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,ƀƀƀƀABC
```

then J = 'ƀƀƀ'

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

You may accomplish the formatted transfer of alphanumeric data 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 unit 6 and stored in the variable V.

The A descriptor deals with variables containing left-justified, blank-filled characters; 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 (m=10); all other variables contain 5 (m=5).

A Descriptor

1. INPUT, where w ⩾ m -- The rightmost m characters of the field are read in and stored left-justified and blank-filled in the associated variable.

2. INPUT, where w < m -- All w characters are read in and stored left-justified and blank-filled in the associated variable.

3. OUTPUT, where w ⩾ m -- m characters are output and right-justified in the field. The remainder of the field is blank-filled.

4. OUTPUT, where w < m -- The left most w characters of the associated variable are output.

R Descriptor

1. 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.

2. INPUT, where w < m -- All w characters are read in and stored right-justified, zero-filled in the associated variable.

3. OUTPUT, where w ⩾ m -- m characters are output and right justified in the field. The remainder of the field is blank filled.

4. OUTPUT, where w < m -- The right most w characters of the associated variable are output.

13.2.8 Transferring Alphanumeric Data

You may transmit alphanumeric data 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, you may use the following statement sequence 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 that 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 rightmost characters are lost.

Examples

```
         WRITE (1,101)
     101 FORMAT (17Hb̸PROGRAMb̸COMPLETE)
```

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 (17Hb̸PROGRAMb̸COMPLETE)
```

would change the contents of statement 101 to

```
     101 FORMAT (17HSTARTb̸b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸)
```

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, you may use the descriptors

```
     101 FORMAT (17Hb̸PROGRAMb̸COMPLETE)
```

and

```
     101 FORMAT ('b̸PROGRAMb̸COMPLETE')
```

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 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

You may place an alphanumeric field descriptor among other fields of the format. For example, you may use the statement:

```
FORMAT (I4,7HƀFORCE=F10.5)
```

to output the line:

```
ƀƀ22ƀFORCE=ƀƀ17.68901
```

You may omit the separating comma after an alphanumeric format field, as shown in the foregoing statement.

When you omit a comma delimiter 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, use a slash to indicate a new record. For example, the statement

```
FORMAT (3O8/I5,2F8.4)
```

is equivalent to

```
FORMAT (3O8)
```

for the first record, and

```
FORMAT (I5,2F8.4)
```

for the second record.

You may omit separating commas when you use a slash. 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:

1. that group repeat specification terminated by the last right parenthesis of the next lower level group, or

2. level zero if no higher level group exists.

Thus, the statement

```
FORMAT (F7.2,(2(E15.5,E15.4),I7))
level 0                      level 0
       level 1    level 1
              level 2
```

causes the format

```
2(E15.5,E15.4),I7
```

to be used after 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 the next 5 records have the format

```
2(E15.5,E15.4),I7
```

13.2.11 Record Formatting Field Descriptors

You may use two field descriptors, Tw and nX, to position data within a record.

You may use the field descriptor Tw 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 that specifies the character position, in a 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 1X, except where a forms control character is used.

NOTE

Two successive T field specifications will result in the second field overwriting the first field if the fields overlap.

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

```
ABCb̸b̸b̸XYZ
```

is read with the format specification

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read in that order.

You may use the field descriptor nX 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 (5Hb̸STEP,I5,10X,2HY=,F7.3)
```

may be used to print the line

```
STEPb̸b̸b̸28b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸Y=b̸-3.872
```

13.2.12 $ Format Descriptor

A $ format descriptor at the end of an output FORMAT is used to suppress the carriage return at the end of the current record. It is mainly used on terminal output but will work on non-terminal devices. A $ format descriptor is ignored in input FORMATs and has no effect if embedded in an output FORMAT. The $ format descriptor must be the next format descriptor to be processed when the corresponding output list is exhausted for the $ descriptor to have the defined effect.

13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS

You may use the first character of an ASCII record to control the spacing operations of the line printer or Teletype terminal printer unit on which the record is being printed. Specify the control character desired by beginning the FORMAT field specification for the ASCII record to be output with 1Ha...where a is the desired control character. Table 13-5 describes the control characters permitted in FORTRAN-20 and the effect each has on the printing device.

Table 13-5
FORTRAN-20 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.

CHAPTER 14

DEVICE CONTROL STATEMENTS

14.1 INTRODUCTION

You may use the following device control statements in FORTRAN-20 source programs:

1. REWIND
2. UNLOAD
3. BACKSPACE(1)
4. ENDFILE
5. SKIPRECORD(1)
6. SKIPFILE
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 devices (MTA). In FORTRAN-20, however, the device control operations are simulated for disk devices.

The following paragraphs describe the form and use of the device control statements.

14.2 REWIND STATEMENT

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

1. The results of these commands are unpredictable when used on list-directed and NAMELIST-controlled data.

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

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

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

You cannot use this statement for files set up for random access, list-directed, or NAMELIST-controlled I/O operations.

Example: BACKSPACE 16

14.5 END FILE STATEMENT

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 that 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

Form: SKIP RECORD u

Use: In accessing the file located on device u, skip the record immediately following the current (last accessed) record.

NOTE

You cannot use this statement for files set up for random access, list-directed, or NAMELIST-controlled I/O operations.

Example: SKIP RECORD 16

14.7 SKIP FILE STATEMENT

Form: SKIP FILE u

Use: In accessing the medium located on unit u, skip the file immediately following the current (last accessed) file. If there is no file after the current file, an error will occur.

Example: SKIP FILE 01

14.8 BACKFILE STATEMENT

Form: BACKFILE u

Use: Move the medium mounted on device u to the start of the file that precedes the current (last accessed) file.

If there is no file before the current file, 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

Table 14-1 summarizes the form and use of device control statements.

Table 14-1
Summary of FORTRAN-20 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 into the current file
SKIP RECORD u	Skip the next record
SKIP FILE u	Skip the next file
BACKFILE u	Move medium backwards one file
BACKSPACE u	Move medium back one record

CHAPTER 15

SUBPROGRAM STATEMENTS

15.1 INTRODUCTION

Procedures you use repeatedly in a program may be written once and then referenced each time you need the procedure. 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 procedures that may be referenced are:

1. statement functions,
2. intrinsic functions (FORTRAN-defined functions),
3. external functions, and
4. subroutines.

The first three of the foregoing categories are referred to collectively as functions or function procedures; procedures of the last category are referred to as subroutines or subroutine procedures.

15.1.1 Dummy and Actual Arguments

Since you may reference subprograms 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 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 argument in the argument list of the subprogram reference.

Dummy arguments may be:

1. variables,
2. array names,
3. subroutine identifiers,
4. function identifiers, or
5. statement label identifiers that are denoted by the symbol "*", "$", or "&".

When you reference a subprogram, 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:

1. 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.
2. Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.
3. A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding argument.
4. 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.
5. A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.
6. A dummy argument representing an external function must have an external function as its actual argument.
7. 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 name you assign that consists of one to six characters. The name you use must conform to the rules for symbolic names given in Section 3.3. The type of a statement function is determined either by the first character of its name or by it being explicitly declared in a 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 that may use the given dummy arguments and indicates how they are combined to obtain the desired value. You may use the dummy arguments as variables or indirect function references; but you cannot use them 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 FORTRAN-defined functions (Section 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 that it contains. A statement function cannot reference itself.

You must define all statement functions within a program unit 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 (FORTRAN DEFINED FUNCTIONS)

Intrinsic functions are subprograms supplied by FORTRAN. Reference an intrinsic function by using its name as an operand in an expression. The name always refers to the intrinsic function unless it is preceded by an asterisk (*) or ampersand (&) in an EXTERNAL statement, declared in a conflicting explicit type statement, or specified as a routine dummy parameter.

Table 15-1 describes FORTRAN-20 intrinsic functions and their arguments. Notice that octal constants are not allowed as arguments.

Table 15-1
Intrinsic Functions (FORTRAN-20 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 ≤ arg	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 + i*Arg	2	Real	Complex
Truncation:					
Real to real	AINT	Sign of arg* largest integer ≤ arg	1	Real	Real
Real to integer	INT*		1	Real	Integer
Double to integer	IDINT		1	Double	Integer

* In line functions.

Table 15-1 (Cont.)
Intrinsic Functions (FORTRAN-20 Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
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(Arg1,Arg2,...)	>1	Integer	Real
	AMAX1*		>1	Real	Real
	MAX0*		>1	Integer	Integer
	MAX1		>1	Real	Integer
	DMAX1		>1	Double	Double
Minimum Value:					
	AMIN0	Min(Arg1,Arg2,...)	>1	Integer	Real
	AMIN1*		>1	Real	Real
	MIN0*		>1	Integer	Integer
	MIN1		>1	Real	Integer
	DMIN1		>1	Double	Double
Transfer of Sign:					
Real	SIGN*	Sign(Arg2)* Arg1	2	Real	Real
Integer	ISIGN		2	Integer	Integer
Double precision	DSIGN		2	Double	Double
Positive Difference:					
Real	DIM*	Arg1-Min(Arg1,Arg2)	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 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 you assign to the function. The name may consist of from one to six characters, the first of which must be alphabetic. You may include the optional size modifier (*n) with the name if the type is specified. (Refer to Section 6.3.)
(arg1,...,argn)	is a list of dummy arguments.

If you omit type 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 with the subprogram itself.

Note that if you want to use the same name for a user-defined function and the name of a FORTRAN-20 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:

1. You must use the symbolic name assigned a FUNCTION subprogram as a variable name in the subprogram. During each execution of the subprogram, this variable must be defined and, once defined, may be referenced or 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 Section 15.6 for a description of this statement.

2. You may not use the symbolic name of a FUNCTION subprogram in any nonexecutable statement in the subprogram except in the initial FUNCTION statement or a type statement.

3. Dummy argument names may not appear in any EQUIVALENCE, COMMON, or DATA statement used within the subprogram.

4. 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.

5. 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.

6. 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 (FORTRAN-20 Defined Functions)

FORTRAN-20 contains a group of predefined external functions that are called 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 FORTRAN-20 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 that was specified unprefixed in an EXTERNAL statement is used as a routine parameter, it is assumed to reference a FORTRAN-20 defined function of the same name, or if none exists, 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 by use of an EXTERNAL statement.

15.5 SUBROUTINE SUBPROGRAMS

A subroutine is an external computational procedure that 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 (FORTRAN-20 Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Exponential:					
Real	EXP	{Arg}	1	Real	Real
Double	DEXP		1	Double	Double
Complex	CEXP		1	Complex	Complex
Logarithm:					
Real	ALOG	ln (Arg)	1	Real	Real
	ALOG10	log (Arg)	1	Real	Real
Double	DLOG	ln (Arg)	1	Double	Double
	DLOG10	log (Arg)	1	Double	Double
Complex	CLOG	ln (Arg)	1	Complex	Complex
Square Root:					
Real	SQRT*	(Arg)**1/2	1	Real	Real
Double	DSQRT	(Arg)**1/2	1	Double	Double
Complex	CSQRT	(Arg)**1/2	1	Complex	Complex
Sine:					
Real (radians)	SIN*		1	Real	Real
Real (degrees)	SIND		1	Real	Real
Double (radians)	DSIN	{sin(Arg)}	1	Double	Double
Complex	CSIN		1	Complex	Complex
Cosine:					
Real (radians)	COS*		1	Real	Real
Real (degrees)	COSD		1	Real	Real
Double (radians)	DCOS	{cos(Arg)}	1	Double	Double
Complex	CCOS		1	Complex	Complex

*Generic functions

Table 15-2 (Cont.)
Basic External Functions (FORTRAN-20 Defined Functions)

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
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*	atan(Arg1/Arg2)	2	Real	Real
Two DOUBLE arguments	DATAN2	atan(Arg1/Arg2)	2	Double	Double
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

*Generic functions

The following rules control the structuring of a subroutine subprogram:

1. You may not use the symbolic name of the subprogram in any statement within the defined subprogram except the SUBROUTINE statement itself.

2. You may not use the given dummy arguments in an EQUIVALENCE, COMMON, or DATA statement within the subprogram.

3. The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

4. 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.

5. Dummy arguments that represent statement labels may be either an *, $, or &.

6. 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.

7. Subroutine subprograms may have as many entry points as desired (see description of ENTRY statement given in Section 15.7).

15.5.1 Referencing Subroutines (CALL Statement)

You must reference subroutine subprograms by 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:

1. a constant

2. a variable name

3. an array element identifier

4. an array name

5. an expression

6. the name of an external subroutine, or

7. 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 FORTRAN-20 Supplied Subroutines

FORTRAN-20 provides you with an extensive group of predefined subroutines. Table 15-3 gives the descriptions and names of these predefined subroutines.

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 that follows the statement that called the subprogram.

The multiple returns form of this statement, i.e., RETURN e, enables you 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 that 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.

You may use any number of RETURN (standard return) statements 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 Subprogram

Reference an external function subprogram 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:

1. a variable name,
2. an array element identifier,
3. an array name,
4. an expression,
5. a statement number, or

6. 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(IX,IY,IZ)
.
.
.
RETURN
END
```

may be referenced in the following manner:

```
.
.
TOTAL=ICALC(IAA,IAB,IAC)+500
```

15.7 MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT)

FORTRAN-20 provides an ENTRY statement that enables you to specify additional entry points into an external subprogram. This statement used in conjunction with a RETURN statement enables you to employ only one computational routine of a subprogram that 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

1. variable names,

2. array declarators,

3. the name of an external procedure (SUBROUTINE or FUNCTION), or

4. statement label identifiers that are denoted by either a *, $, or & symbol.

The rules for the use of an ENTRY statement follow:

1. The ENTRY statement allows entry into a subprogram at a place other than that defined by the subroutine or function statement. You may include any number of ENTRY statements in an external subprogram.

2. Execution is begun at the first executable statement following the ENTRY statement.

3. Appearance of an ENTRY statement in a subprogram does not negate the rule that statement functions in subprograms must precede the first executable statement.

4. Entry statements are nonexecutable and do not affect the execution flow of a subprogram.

5. You may not use an ENTRY statement in a main program or have a subprogram reference itself through its entry points.

6. You may not use an ENTRY statement in the range of a DO or an extended DO statement construction.

7. The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in SUBROUTINE or FUNCTION statements of 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.

8. Entry into a subprogram initializes only the dummy arguments of the referenced ENTRY statement.

9. You may not reference a dummy argument unless it appears in the dummy list of an ENTRY, SUBROUTINE, or FUNCTION statement by which the subprogram is entered.

10. The source subprogram must be ordered such that references to dummy arguments in executable statements follow the appearance of the dummy argument in the dummy list of a SUBROUTINE, FUNCTION, or ENTRY statement.

11. Dummy arguments that were defined for a subprogram by some previous reference to the subprogram are undefined for subsequent entry into the subprogram.

12. The value of the function must be returned by use of the current entry name.

Table 15-3
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
AXIS	CALL AXIS(X,Y,ASC,NASC,S,THETA,XMIN,DX) AXIS causes an axis with tick marks and scale values at 1-inch increments to be drawn. An identifying label may also be plotted along the axis. Parameters X and Y specify the start of the axis. The axis is plotted, starting at X, Y, at an angle of THETA degrees for a distance of S inches. The angle THETA is usually either 0 (X axis) or 90.0 (Y axis). Characters ASC of array ASC are plotted as a label for the axis drawn. If NASC is positive, the tic marks, label, and scale values are placed on the counterclockwise side of the axis; if NASC is negative, the foregoing items are placed on the clockwise side of the axis. Parameter XMIN is the value of the scale at the beginning of the axis; parameter DX is the change in scale for a 1-inch increment. The values of XMIN and DX may be determined by subroutine SCALE.
DATE	CALL DATE (array) This subroutine places today's date as left-justified ASCII characters into a dimensioned 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-letter month abbreviation, 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	CALL DEFINE FILE (u,s,v,f,pj,pg) The arguments of this subroutine are defined as follows: u = logical device numbers. s = the size of the records comprising 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 modified 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.

Table 15-3 (Cont.)
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
	f = filename to be given the file being defined. pj = your project number. pg = your programmer's number. NOTE Numbers pj and pg identify your File Directory. Example The statement CALL DEFINE FILE (1,10,ASCVAR,'FORTFL.DAT',0,0) establishes a file named FORTFL.DAT on device 01, a disk, which contains ten word records. The associated variable is ASCVAR, and the file is in your area. 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.
DUMP	CALL DUMP (L(1),U(1),F(1),...,L(n),U(n),F(n)) DUMP causes particular portions of memory to be dumped. L(1) and U(1) are the variable names that give the limits of 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.

Table 15-3 (Cont.)
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
ERRSET	CALL ERRSET(N) ERRSET allows you to control the typeout of execution-time arithmetic error messages. ERRSET is called with one integer argument. Typeout of all arithmetic and library error messages is suppressed after N occurrences of these error messages. If ERRSET is not called, the default value of N is 2.
ERRSNS	CALL ERRSNS(I,J) ERRSNS allows you to determine the exact nature of an error on READ, WRITE, OPEN, or CLOSE that was trapped with the "ERR= statement label" option. ERRSNS returns one or two integer values that describe the status of the last I/O operation performed by FOROTS. (The second integer value is optional.) CALL ERRSNS(I,J) returns a FORTRAN-standardized number in I and a processor-dependent number in J to describe the last I/O operation. See Appendix H and Table H-1 for more information and a detailed description of the values returned.
EXIT	EXIT EXIT returns control to the Monitor and, therefore, terminates the execution of the program.
ILL	CALL ILL 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.
LEGAL	CALL LEGAL 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.
LINE	CALL LINE (X,Y,N,K) LINE causes a line to be drawn through the N points specified by (X(1),Y(1)),(X(2),Y(2))...(X(N),Y(N)) where the elements of X and Y are spaced K words apart in storage.

Table 15-3 (Cont.)
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
MKTBL	CALL MKTBL(I,J) MKTBL specifies a special character set where I is the number to be assigned the set and J contains the starting address of a character table of 200(8) consecutive words. In each character table word, the left half contains the number of strokes in the character (0 if nothing is to be plotted for the word) and the right half contains the address of the table of strokes for the character.
NUMBER	CALL NUMBER(X,Y,SIZE,FNUM,THETA,NDIGIT) NUMBER causes a floating- point number to be plotted as text. Parameters X, Y, SIZE, and THETA have the same meanings as for the SYMBOL call. Parameter NDIGIT is the number of digits plotted to the right of the decimal point. If NDIGIT is a negative value, only the integer part of the number is plotted. FNUM specifies the number to be plotted.
PDUMP	CALL PDUMP(L(1),U(1),F(1),...,L(n),U(n),F(n) The arguments of PDUMP are the same as those of DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.
PLOT	CALL PLOT(X,Y,IPEN) PLOT moves the pen in a straight line from its current position to the position specified by X,Y. If IPEN=3, the pen is raised before the movement; if IPEN=2 the pen is lowered before movement; if IPEN=1 the pen is left unchanged from its previous state. If the value of IPEN is negative (-1, -2 or -3) the pen action is the same as for the corresponding positive values except that on completion of the indicated motion, the new pen position is taken as a new origin and the output buffer is sent to the plotter. The plotter is not released on completion of the specified movement.
PLOTS	CALL PLOTS (I) PLOTS is the plotter setup routine. If the plotter is not available, I is set to -1; if it is available, I is set to 0. This call must be the first plotter routine called.

Table 15-3 (Cont.)
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
RELEAS	CALL RELEAS(unit) RELEAS closes out I/O on a device initialized by the FORTRAN Object Time System and returns it to the uninitialized state. RELEAS should be the last call referencing that device.
SAVRAN	CALL SAVRAN(I) SAVRAN is called with one integer argument. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SCALE	CALL SCALE(X,N,S,XMIN,DX) SCALE selects scale values for an AXIS call where X and N specify a 1-dimensional array X with the length N. Parameter S specifies the length of the desired axis, SCALE determines a value of DX that allows X to be plotted in S inches. XMIN is selected as the smallest element of the array X, and is truncated to be a multiple of DX.
SETABL	CALL SETABL(I,J) SETABL specifies a character set where I is an integer that 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	CALL SETRAN (I) SETRAN has one argument, which must be a non-negative integer <2**(31). The starting value of the function RAN is set to the value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.
SORT	CALL SORT ('OUTPUT/SWS=INPUT/SWS,INPUT/SWS') SORT sorts one or more files using the SORT program. The argument is an ASCII string that represents (version 3 or later) the standard SORT command string. Its components are: OUTPUT = file specification of the output file. INPUT = file specification of the input file(s). SWS = one or more switches for the output file, the input file(s), the sorting process, and sometimes SCAN. The switches not allowed in the FORTRAN call are:/BLOCK, /COMP3, /EBCDIC, /INDUSTRY, /LABEL, /SIXBIT, and /VERSION.

Table 15-3 (Cont.)
FORTRAN-20 Library Subroutines

Subroutine Name	Effect
	Wild card format is not allowed in the SORT call. For information about using the SORT program, see the SORT User's Guide. Example: `CALL SORT('SRTFIL.SRT=INSTRT/REC:80/KEY:1:2')`
SYMBOL	`CALL SYMBOL(X,Y,SIZE,ASC,THETA,NASC)` SYMBOL raises the plotter pen and moves it to position specified by X and Y. Lower pen and plot characters found in array ASC. Parameter SIZE specifies the height of the characters plotted in inches (floating- point values); THETA specifies the direction of the base line on which the text of array ASC is to be plotted, and NASC specifies the number of characters in array ASC.
TIME	`CALL TIME(X)` or `CALL TIME(X,Y)` 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 in the same form as the one-argument call, and the second has the form `bss.t` where b is a blank, ss is in seconds, and t is in tenths of a second.
WHERE	`CALL WHERE(X,Y)` Variables X and Y are set to the values which identify the current position of the pen.

CHAPTER 16

BLOCK DATA SUBPROGRAMS

16.1 INTRODUCTION

Use block data subprograms to initialize data to be stored in any common areas. You may use only specification and DATA statements, i.e., DATA, COMMON, DIMENSION, EQUIVALENCE, and TYPE, in BLOCK DATA subprograms. A subprogram of this type must start with a BLOCK DATA statement.

You may enter initial values 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	b/	!	"	#	$	%	&	'
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 (04x–13x), 95 (04x–17x)

Characters inside parentheses are ASCII-1963 Standard.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
		DEL	Delete (Rubout)

APPENDIX B

USING THE COMPILER

This appendix explains how to access FORTRAN-20 and how to make use of the information it provides. You should be familiar with the FORTRAN-20 language and the DECsystem-20 TOPS-20 monitor.

B.1 RUNNING THE COMPILER

The command to run FORTRAN-20 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)
```

You are given the following options:

1. The filenames can be fully specified with SFD paths.
2. You 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.
3. Program units need not be terminated at file boundaries and may consist of more than one file.
4. If no object filename is specified, no relocatable binary file is generated.
5. If no listing filename is specified, no listing is generated.
6. If no extension is given, the defaults are .LST (listing), .REL (relocatable binary), and .FOR (source) for their respective files.

B.1.1 Switches Available with FORTRAN-20

Switches to FORTRAN-20 are accepted anywhere in the command string. They are totally position- and file-independent. Table B-1 lists the switches.

Table B-1
FORTRAN-20 Compiler Switches

Switch	Meaning	Defaults
CROSSREF	Generates a file that can be input to the CREF program	OFF
DEBUG	(See Section B.1.1.1.)	OFF
EXPAND	Includes the octal-formatted version of the object file in the listing.	OFF
INCLUDE	Compiles a D in column 1 as space.	OFF
LNMAP	Produces a line number/octal location map in the listing only if /MACROCODE was not specified.	OFF
MACROCODE	Adds the mnemonic translation of the object code to the listing file.	OFF
NOERRORS	Does not print error messages on the terminal.	OFF
NOWARNINGS	Does not output warning messages.	OFF
OPTIMIZE	Performs global optimization.	OFF
SYNTAX	Performs syntax check only.	OFF

Each switch must be preceded by a slash (/). Switch names need only contain those letters that are required to make the switch name unique. You 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.FOR and S2FILE.FOR to appear in LFILE.LST.

All switches used or implied are printed at the top of each listing page.

B.1.1.1 **The /DEBUG Switch** - The /DEBUG switch tells FORTRAN-20 to compile a series of debugging features into your program. Several of these features are specifically designed to be used with FORDDT. Refer to Appendix E for more information. By adding the modifiers listed in Table B-2, you can include specific debugging features.

Table B-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. (You may use this option without FORDDT.)
:NONE	Do not include any debug features.
:ALL	Enable all debugging aids.

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 you wish all but one or two modifiers and do not wish to list them all, you may use the prefix "NO" before the switch you wish 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 you include more than one statement 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 as a result of the addition of various /DEBUG options.

:DIMENSIONS	For each array, 3+3*N words where N is the number of dimensions, and up to three 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 you do not specify :BOUNDS, approximately 1+3*(N-1) words will be used.

B.1.2 LOAD-Class Commands

You can invoke FORTRAN-20 by using LOAD-class commands. These commands cause the monitor to interpret the command and construct new command strings for the system programs actually processing the command.

COMPILE
LOAD
EXECUTE
DEBUG

Example

```
.EXEC ROTOR
```

The compiler switches OPT, CREF, and DEBUG may be specified in LOAD-class commands and may be used globally or locally.

Example

```
.EXECUTE/CREF P1.FOR,P2.FOR/DEBUG:NOBOU
```

The other compiler switches must be passed in parentheses for each specific source file.

Example

```
.EXECUTE P1.FOR(M,I)
```

Refer to the Monitor Calls User's Guide for further information.

B.2 READING THE LISTING

When you request 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 blocks and the relative locations (in octal) of the variables in each COMMON block.

4. A listing of all equivalenced variables or arrays and their relative locations.

5. A listing of the subprograms referenced (both user defined and 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 you used the /MACRO switch, a mnemonic printout of the generated code (in a format similar to MACRO) 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 instruction 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.

If you used the /LNMAP switch and did NOT use the /MACRO switch, a line number/octal location map is appended to the listing. This section lists the line numbers in increments of 10 on subsequent lines and each number from 0 through 9 for each line in adjacent columns. The numbers appearing inside the matrix are the relative octal locations of the statements in the FORTRAN program unit. For example, to find the relative octal location of line number 001043, find the row marked 001040 and then column 3 on that line. The number in that place is the desired relative location. This listing can be very large and sparse for line-numbered files with large increments, such as those produced by EDIT.

NOTE

One FORTRAN line can produce multiple octal locations. In this case the line number map lists only the first location.

9. A list of all argument blocks 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 compilations.

B.2.1 Compiler Generated Variables

In certain situations the compiler will generate internal variables. Knowing what these variables represent can help you read the macro expansion. The variables are of the form:

.letter digit digit digit digit

i.e., .S0001

where:

Letter	Function of Variable
A	Register save area.
F	Arithmetic statement function formal parameters.
I	Result of a DO LOOP initial value expression or parameter of an adjustably dimensioned array.
O	Result of a common subexpression (see Section C.2.1.1) or constant computation (C.2.1.3).
Q	Temporary storage for expression values.
R	Result of reduced operator strength expression (C.2.1.2).
S	Result of the DO LOOP step size expression of computed iteration count for a loop.

You may find these variables on the listing under SCALARS and ARRAYS.

The following example shows a listing where all these features are pointed out.

Program Name | Source Filename | Compiler Version | Compiled for KI processor | MACRO Code equivalent included

```
MAIN.     TIM1     FORTRAN V.5(515) /KI/M  16-MAR-77       16:05   PAGE 1

                   IMPLICIT INTEGER (A-Z)
                   DIMENSION A(100,200),B(100,200)
                   SUM1=0
                   SUM2=0
                   DO 100 J=1,200
                   DO 100 I=1,100
                   K1=I*J
                   IF(K1.LT.500.OR.K1.GT.1500) K1=0
                   A(I,J)=K1
                   K2=I+J
                   IF(K2.EQ.100.OR.K2.EQ.200.OR.K2.EQ.300) K2=K2+1
                   B(I,J)=K2
                   SUM1=SUM1+K1
                   SUM2=SUM2+K2
          100      CONTINUE
          C
                   TYPE 10,SUM1,SUM2
          10       FORMAT(7H SUM1= ,I9,10H    SUM2= ,I9)
                   END
```

```
SUBPROGRAMS CALLED


SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]
                                                                  ┌─Compiler Generated Variable
*K1     1       B       2       *J      47042   A       47043   .S0001  116103
 .S0000 116104  *SUM2   116105  *I      116106  *K2     116107  *SUM1   116110
                                                   Internal sequence number of first
┌──────────────────────────────────────────────────instruction that goes with this line
LINE    LOC     LABEL   GENERATED CODE
        ▲──────────────────────────────────────────Octal displacement of instruction
        0               JFCL    0,0
        1               JSP     16,RESET.
        2                       0,0
3       3               SETZB   2,SUM1
4       4               MOVEM   2,SUM2
5       5               MOVE    2,[777470000001]
        6               HLREM   2,.S0000
        7       2M:
                        HRRZM   2,J
6       10      3M:
                        MOVE    2,[777634000001]
7       11      4M:
                        MOVE    3,J
        12              IMULI   3,0(2)
        13              MOVEM   3,K1
8       14              CAIL    3,764
        15              CAILE   3,2734
        16              JRST    0,6M
        17              JRST    0,5M              Compiler
8       20      6M: ◄─────────────────────────────Generated
                        SETZB   4,K1              Label
```

```
MAIN.     TIM1     FORTRAN V.5(515) /KI/M  16-MAR-77        16:05    PAGE 1-1

9         21       5M:
                             MOVEI    3,144
          22                 IMUL     3,J
          23                 ADDI     3,0(2)
          24                 MOVE     4,K1
          25                 MOVEM    4,A-145(3)
10        26                 MOVE     3,J
          27                 ADDI     3,0(2)
          30                 MOVEM    3,K2
11        31                 MOVE     5,K2
          32                 CAIE     5,144
          33                 CAIN     5,310
          34                 JRST     0,8M
          35       9M:
                             CAIN     5,454
11        36       8M:
                             AOS      3,K2
12        37       7M:
                             MOVEI    3,144
          40                 IMUL     3,J
          41                 ADDI     3,0(2)
          42                 MOVE     5,K2
          43                 MOVEM    5,B-145(3)
13        44                 ADDM     4,SUM1
14        45                 ADDM     5,SUM2
15        46       100P:<-------------------------Program label
                             AOBJN    2,4M
          47                 AOS      2,J
          50                 AOSGE    0,.S0000
          51                 JRST     0,3M
17        52                 MOVEI    16,10M
          53                 PUSHJ    17,OUT.
          54                 MOVEI    16,11M
          55                 PUSHJ    17,IOLST.
19        56                 MOVEI    16,1M
          57                 PUSHJ    17,EXIT.
```

```
ARGUMENT BLOCKS:


        60              0,,0
        61      1M:     0,,0
        62              777773,,0
        63      10M:    0,,777777
        64              0,,0
        65              0,,0
        66              340,,10P
        67              0,,7
        70              0,,0
        71      11M:    1100,,SUM1
        72              1100,,SUM2
        73              4000,,0


MAIN.   TIM1    FORTRAN V.5(515) /KI/M  16-MAR-77       16:05   PAGE 1-2


FORMAT STATEMENTS (IN LOW SEGMENT):

18      116111  10P:    (7H S
        116112          UM1=
        116113          ,I9,1
        116114          0H
        116115           SUM2
        116116          = ,I9
        116117          )

MAIN.   [ NO ERRORS DETECTED ]
```

```
MAIN.     TIM1     FORTRAN V.5(515) /KI     16-MAR-77        16:09     PAGE 1

                   IMPLICIT INTEGER (A-Z)
                   DIMENSION A(100,200),B(100,200)
                   SUM1=0
                   SUM2=0
                   DO 100 J=1,200
                   DO 100 I=1,100
                   K1=I*J
                   IF(K1.LT.500.OR.K1.GT.1500) K1=0
                   A(I,J)=K1
                   K2=I+J
                   IF(K2.EQ.100.OR.K2.EQ.200.OR.K2.EQ.300) K2=K2+1
                   B(I,J)=K2
                   SUM1=SUM1+K1
                   SUM2=SUM2+K2
          100      CONTINUE
          C
                   TYPE 10,SUM1,SUM2
          10       FORMAT(7H SUM1= ,I9,10H    SUM2= ,I9)
                   END

SUBPROGRAMS CALLED


SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*K1      1         B        2         *J       47042    A        47043     .S0001 116103
 .S0000 116104    *SUM2     116105    *I       116106   *K2      116107    *SUM1   116110
```

```
LINE NUMBER/OCTAL LOCATION MAP

      : 0        1        2        3        4        5        6        7        8        9
------:------------------------------------------------------------------------------------------
      :
00000 :                            3        4        5        10       11       14       21
00010 : 26       31       37       44       45       46                52                56
MAIN.   [ NO ERRORS DETECTED ]
```

Line number 11 starts at location 31.
The previous listing shows that line 11
uses locations 31 through 36, but only
the first location is shown here.

```
MAIN.   TIM1    FORTRAN V.5(515) /KI/OPT/M      16-MAR-77       16:07   PAGE 1


00001           IMPLICIT INTEGER (A-Z)
00002           DIMENSION A(100,200),B(100,200)
00003           SUM1=0
00004           SUM2=0
00005           DO 100 J=1,200
00006           DO 100 I=1,100
00007           K1=I*J
00008           IF(K1.LT.500.OR.K1.GT.1500) K1=0
00009           A(I,J)=K1
00010           K2=I+J
00011           IF(K2.EQ.100.OR.K2.EQ.200.OR.K2.EQ.300) K2=K2+1
00012           B(I,J)=K2
00013           SUM1=SUM1+K1
00014           SUM2=SUM2+K2
00015   100     CONTINUE
00016   C
00017           TYPE 10,SUM1,SUM2
00018   10      FORMAT(7H SUM1= ,I9,10H    SUM2= ,I9)
00019           END
```

```
SUBPROGRAMS CALLED


SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*K1      1        B       2       .R0001 47042    .R0000 47043   *J      47044
 A       47045   .S0001  116105   .S0000 116106   *SUM2  116107  *I      116110
 .O0001  116111  *K2     116112   *SUM1  116113

LINE     LOC      LABEL   GENERATED CODE

         0                JFCL    0,0
         1                JSP     16,RESET.
         2                        0,0
 4       3                SETZB   10,11
 *       4                MOVEI   12,144
         5                MOVEM   12,.R0001
 5       6                MOVNI   12,310
         7                MOVEI   7,1
         10               MOVEM   12,.S0000
 *       11       4M:
                          MOVE    6,7
 6       12               MOVE    2,[777634000001]
 *       13       5M:
                          MOVEI   4,0(2)
         14               ADD     4,.R0001
 7       15               MOVE    5,6
 8       16               CAIL    5,764
         17               CAILE   5,2734
         20               JRST    0,7M
         21               JRST    0,6M
 8       22       7M:
```

Optimizer Created Variables

Optimizer Created Statements

```
MAIN.     TIM1      FORTRAN V.5(515) /KI/OPT/M      16-MAR-77       16:07    PAGE 1-1

                              MOVEI    5,0
9         23        6M:
                              MOVEM    5,A-145(4)
10        24                  MOVE     3,7
          25                  ADDI     3,0(2)
11        26                  CAIE     3,144
          27                  CAIN     3,310
          30                  JRST     0,9M
          31        10M:
                              CAIN     3,454
11        32        9M:
                              ADDI     3,1
12        33        8M:
                              MOVEM    3,B-145(4)
13        34                  ADD      11,5
14        35                  ADD      10,3
*         36                  ADD      6,7 <-
15        37        100P:
                              AOBJN    2,5M
*         40                  MOVEI    12,144
          41                  ADDM     12,.R0001
*         42        1M:                     <-
                              ADDI     7,1
          43                  AOSGE    0,.S0000
          44                  JRST     0,4M
*         45                  MOVEM    11,SUM1 <-
*         46                  MOVEM    10,SUM2 <-
*         47                  MOVEM    5,K1 <-
*         50                  MOVEM    3,K2 <-
17        51                  MOVEI    16,11M
          52                  PUSHJ    17,OUT.
*         53                  MOVEI    16,12M <-
          54                  PUSHJ    17,IOLST.
19        55        2M:
                              MOVEI    16,3M
          56                  PUSHJ    17,EXIT.
```

Optimizer Created Statements

```
ARGUMENT BLOCKS:

        57          0,,0
        60   3M:    0,,0
        61          777773,,0
        62   11M:   0,,777777
        63          0,,0
        64          0,,0
        65          340,,10P
        66          0,,7
        67          0,,0
        70   12M:   1100,,SUM1
        71          1100,,SUM2
        72          4000,,0


MAIN.   TIM1    FORTRAN V.5(515) /KI/OPT/M     16-MAR-77      16:07   PAGE 1-2


FORMAT STATEMENTS (IN LOW SEGMENT):

18      116114  10P:    (7H S
        116115          UM1=
        116116          ,I9,1
        116117          0H
        116120           SUM2
        116121          = ,I9
        116122          )

MAIN.   [ NO ERRORS DETECTED ]
```

B.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 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 your terminal without the statement in error, but they may reference its internal sequence number.

B.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. 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 your 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.

B.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.) Appendix G is a complete list of fatal errors and warning messages.

B.4 CREATING A REENTRANT FORTRAN PROGRAM WITH LINK

To produce a sharable program from the .REL file, such as MAIN.REL, give one of the following commands to LINK:

1. /SEG:DEFAULT MAIN/G

2. /OTS:SHAR MAIN/G

The resulting core image can be SSAVEd or the /SSAVE switch can be used to produce a .SHR file.

APPENDIX C

WRITING USER PROGRAMS

This appendix is a guide for writing effective programs with FORTRAN-20. It contains techniques for optimization, interaction with non-FORTRAN programs, and other useful programming hints.

C.1 GENERAL PROGRAMMING CONSIDERATIONS

The following paragraphs describe programming considerations you should observe when preparing a FORTRAN program to be compiled by FORTRAN-20.

C.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. You must be careful when testing the value of a number within the specified range since, although numbers up to 10**38 may be represented, FORTRAN-20 can only test numbers of up to eight significant digits (REAL precision) and 16 significant digits (DOUBLE precision).

You must also be careful 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 will cause tests within statements, i.e., an arithmetic IF, to fail.

C.1.2 Writing FORTRAN-20 Programs for Execution On Non-DEC Machines

If you prepare a program to run on both a DECsystem-20 computer and a non-DIGITAL machine, you should:

1. Avoid using the non-ANSI standard features of FORTRAN-20, and

2. Consider the accuracy and size of the numbers that the non-DIGITAL machine is capable of handling.

C.1.3 Using Floating-Point DO Loops

FORTRAN-20 permits you to employ non-integer single- or double-precision numbers as the parameter variables in a DO statement. This enables you to generate a wider range of values for the DO loop index variables, which may, in turn, be used inside the loop for computations. Be sure to consider the loss of precision that may occur.

C.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=M1,M2,M3

MAX    (1,((M2-M1)/M3)+1)=Number of cycles
```

The values of the parameters M1, M2, M3 may be of any type; however, you must consider the foregoing formula, particularly when using logicals. One pass through each DO loop is always performed EVEN IF THE RESULT OF THE FOREGOING CALCULATION IS LESS THAN OR EQUAL TO ZERO.

C.1.5 Subroutines - Programming Considerations

Consider the following items when preparing and executing subroutines:

1. During execution, no check is made to see if the proper number of parameters was passed.

2. If the number of actual arguments passed to a subroutine is 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 memory. The proper method is to use F1(2.5D00).

NOTE

You are given no notice if any of the situations described in items 1,2,3,4, and 5 occur.

C.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 ensure proper results 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 (see Section C.2.1.1) and generate the following sequence:

```
temp = RL1*RL2
A    = RS1*temp
B    = RS2*temp
```

The computation of temp will cause an overflow.

You should write the program as follows 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.

C.1.7 Dimensioning of Formal Arrays

When you specify an array as a formal parameter to a subprogram unit, you must indicate to the compiler that the parameter is an array. 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 is a common practice.

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
   ```

   The above is a binary read that will read ten words into ARRAY.

   ```
   SUBROUTINE SUB1(A)
   DIMENSION A(1)
   READ(1)A
   ```

   This binary read will cause one 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 one word of the array FMT to be written over with the characters read from the record on unit 1.

When you use 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.

C.2 FORTRAN-20 GLOBAL OPTIMIZATION

You have the option of invoking the global optimizer during compilation. The 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 syntactic 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.

C.2.1 Optimization Techniques

C.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
         .
         .
     DIF((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.

C.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 this 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 you increment MILES on each pass, you can replace the multiply operation 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.

This foregoing situation occurs frequently in subscript calculations that implicitly contain multiplications whenever the size is two or greater.

C.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 consisting 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 greatly decreases 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 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).

C.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.

C.2.1.5 **Removal of Inaccessible Code** - The optimizer detects and eliminates any code within the source program that cannot be accessed. In general, this will not happen since programmers do 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 object program. A warning message is generated for each inaccessible line removed.

C.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 allocation process is designed to minimize the number of MOVE and MOVEM machine instructions that will appear in the most frequently executed portions of the code.

C.2.1.7 **I/O Optimization** - Every effort is made to minimize the number of required calls to 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.

C.2.1.8 **Uninitialized Variable Detection** - A warning message is generated when a scalar variable is referenced before it has received a value.

C.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 TOs out of the loop), the DO loop index is not needed and can be replaced by the reduced variable.

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
```

This occurs frequently in subscript computation.

C.2.2 Improper Function References

Consider this statement:

```
P = F(X) + Q(Y)
```

If:

1. the evaluation of F(X) defines or changes the variables A, B, and C, and

2. the evaluation of Q(Y) defines or changes the values of B, C, and D,

then it is possible that different values of P could result, depending on which function is evaluated first. Let's see how this works. Let's assign some values (to begin with) to A, B, C, and D and define the functions F(X) and Q(Y):

```
Let:
                    F(X):               Q(Y):
     A = 2.             A = 6.              B = 10.
     B = 3.             B = 7.              C = 11.
     C = 4.             C = 8.              D = 12.
     D = 5.             F = D + 9.          Q = A + 13.
```

Now play computer and evaluate P, calling first F(X), then Q(Y). Now re-evaluate P, calling Q(Y) first, then F(X). Notice that you got different values for P because the variables A, B, C, and D changed value depending on the order in which the functions were called. (Our answers were 33 when F(X) was called first and 36 when Q(Y) was called first.)

The ANSI FORTRAN standard prohibits this kind of situation. But the compiler won't catch it unless you mention the affected variables in the function call itself. The compiler depends on strict adherence to this rule. There's a strong possibility that you won't get the results you want if you don't look for situations of this type and avoid them. Your best bet is to define your variables OUTSIDE the function and not change them in the course of the evaluation of the function itself.

C.2.3 Programming Techniques for Effective Optimization

Observe the following recommendations during the coding of a FORTRAN source program. They will improve the effectiveness of the optimizer.

1. Do not use DO loops with an extended range.
2. Specify label lists when using assigned GO TOs.
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.

C.3 INTERACTING WITH NON-FORTRAN PROGRAMS AND FILES

C.3.1 Calling Sequences

The following paragraphs describe the standard procedures for writing subroutine calls.

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 return will be made to the instruction immediately after the PUSHJ 17 instruction.

   e. If you use the FOROTS trace facility, the calling sequence to a routine F must be

```
MOVEI 16,AP
PUSHJ 17,F
```

where AP is the pointer to the argument list. If you use the trace facility, 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 belong to the calling program; they cannot be read by the called subprogram.

   c. FOROTS assumes that it has control of the stack; therefore, you must not create your own stack. The FOROTS stack is initialized by:

      JSP 16,RESET.

C.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 ACs 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 ACs - FORTRAN-20 FUNCTION subprograms preserve ACs 2 through 15; subroutine subprograms do not.

   The design of the called subprogram cannot depend on the contents of any of the ACs being set up by the calling subprogram, except for ACs 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either FORTRAN-20 or COBOL.

C.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 list 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 AC 16. 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
```

C.3.4 Argument Types

Table C-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	Octal	Reserved
7	Label	Procedure address
10	Double real	Not applicable
11	Not applicable	2-word COMP
12	Double Octal	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-20 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.

C.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 subprograms 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 is 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-20 format floating-point number.

bit 0	sign
bits 1-8	excess 128 exponent
bits 9-35	mantissa

5. Type 6 - Octal

   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 precision real

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 represents the real part, and the second 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 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. (See COBOL Program 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.

C.3.6 Converting Existing MACRO Libraries for use with FORTRAN-20

The following simple example illustrates the FORTRAN-20 calling sequence.

```
MAIN.    EX1      FORTRAN V.5(515) /KI/M  16-MAR-77        16:02    PAGE 1

         C        AN EXAMPLE OF A CALL TO A SUBROUTINE WITH A VARIETY OF ARGUMENTS

                  DOUBLE PRECISION DP
                  DIMENSION B(10)

         C        THE ARGUMENTS ARE:
         C                1.  A REAL VARIABLE
         C                2.  AN ARRAY NAME
         C                3.  AN ARRAY ELEMENT
         C                4.  AN INTEGER VARIABLE
         C                5.  A DOUBLE PRECISION VARIABLE
         C                6.  AN OCTAL CONSTANT
         C                7.  A LITERAL

                  CALL SUB1(A,B,B(I),K,DP,"777,'ABC')

                  END


SUBPROGRAMS CALLED

SUB1

SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

 DP      1        *K       3        B       4        *A       16       *I       17
```

```
LINE     LOC      LABEL    GENERATED CODE

         0                 JFCL     0,0
         1                 JSP      16,RESET.
         2                          0,0
15       3                 MOVE     2,I
         4                 MOVEI    2,B-1(2)
         5                 MOVEM    2,.Q0000
         6                 MOVEI    16,2M
         7                 PUSHJ    17,SUB1
17       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,,.Q0000
         20                100,,K
         21                400,,DP
         22                300,,[000000000777]


MAIN.    EX1      FORTRAN V.5(515) /KI/M  16-MAR-77       16:02    PAGE 1-1


         23                740,,[406050320100]
MAIN.    [ NO ERRORS DETECTED ]
```

```
MAIN.    EX1      FORTRAN V.5(515) /KI/M  16-MAR-77        16:02   PAGE 1


                SUBROUTINE SUB1(REAL1,ARYNAM,ARYELM,INT1,DBLPRC,OCT,LIT)
                DOUBLE PRECISION DBLPRC
                DIMENSION ARYNAM(10)

C               AN EXAMPLE OF THE USE AND MODIFICATION OF FORMAL PARAMETERS

                X1=REAL1
                X2=ARYNAM(J)
                X3=ARYELM
                I1=INT1
                X4=DBLPRC
                I2=OCT
                I3=LIT

                REAL1=X1
                ARYNAM(J)=X2
                ARYELM=X3
                INT1=I1
                DBLPRC=CMPLX(X4,0.)
                OCT="55
                LIT='ZYXW'

                RETURN
                END


SUBPROGRAMS CALLED

CMPLX.


SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*LIT     1        *OCT     2        *X4      3        *ARYELM  4        *X3      5
 DBLPRC  6        *I3      10       *REAL1   11       *J       12       *X2      13
*INT1    14       *I2      15       *X1      16       *I1      17        ARYNAM  20
```

```
LINE      LOC       LABEL     GENERATED CODE

          0                   636542,,210000

          SUB1:
2         0                   MOVEM    16,,A0016
          1                   MOVE     0,@0(16)
          2                   MOVEM    0,REAL1
          3                   MOVEI    1,@1(16)
          4                   MOVEM    1,ARYNAM
          5                   MOVE     1,@2(16)
          6                   MOVEM    1,ARYELM
          7                   MOVE     2,@3(16)
          10                  MOVEM    2,INT1
          11                  DMOVE    4,@4(16)
          12                  DMOVEM   4,DBLPRC


SUB1      EX1       FORTRAN V,5(515) /KI/M  16-MAR-77        16:02    PAGE 1-1


          13                  MOVE     3,@5(16)
          14                  MOVEM    3,OCT
          15                  MOVE     6,@6(16)
          16                  MOVEM    6,LIT
8         17        3M:
                              MOVEM    0,X1
9         20                  MOVE     7,J
          21                  ADD      7,ARYNAM
          22                  MOVE     7,777777(7)
          23                  MOVEM    7,X2
10        24                  MOVEM    1,X3
11        25                  MOVEM    2,I1
12        26                  PUSHJ    17,SNG,4
          27                  MOVEM    4,X4
13        30                  FIX      3,3
          31                  MOVEM    3,I2
14        32                  MOVEM    6,I3
16        33                  MOVEM    0,REAL1
```

```
17      34              MOVE    3,J
        35              ADD     3,ARYNAM
        36              MOVEM   7,777777(3)
18      37              MOVEM   1,ARYELM
19      40              MOVEM   2,INT1
20      41              MOVEI   5,0
        42              MOVEI   5,0
        43              DMOVEM  4,DBLPRC
21      44              MOVEI   2,55
        45              MOVEM   2,OCT
22      46              MOVE    2,[552633053500]
        47              MOVEM   2,LIT
25      50      2M:
                        MOVE    16,,A0016
        51              MOVE    0,REAL1
        52              MOVEM   0,@0(16)
        53              MOVE    0,ARYELM
        54              MOVEM   0,@2(16)
        55              MOVE    0,INT1
        56              MOVEM   0,@3(16)
        57              DMOVE   0,DBLPRC
        60              DMOVEM  0,@4(16)
        61              MOVE    0,OCT
        62              MOVEM   0,@5(16)
        63              MOVE    0,LIT
        64              MOVEM   0,@6(16)
        65              POPJ    17,0

 ARGUMENT BLOCKS:


        66              0,,0
        67      1M:     0,,0
SUB1    [ NO ERRORS DETECTED ]
```

To convert existing MACRO programs conveniently so that they will still load and execute correctly when called from FORTRAN-20:

1. Transfer the initial entry sequence for a routine to

```
entry:     CAIA
           PUSH 17,CEXIT.##
```

2. Change all returns to POPJ 17,0

These are the functions performed by the HELLO and GOODBY macros. These macros (available in the file FORPRM.MAC, part of the FOROTS release) were successfully used to convert the library routines to run with FORTRAN-20.

In addition, since the FORTRAN-20 compiler uses the indirect bits on argument lists (note that this permits shared, pure code argument lists), it is essential for code that accesses parameters to take this into account. Specifically, sequences that obtained the values of parameters through use of operations such as

```
HRRZ R,1(16)
```

to pick up the address of the second argument should be changed to

```
MOVEI R,@1(16)
```

The latter operation will work when interfacing with FORTRAN-20.

Refer to the previous example, which illustrates the code generated by the FORTRAN-20 compiler, for specific details of how each argument is accessed. Note that in the case of the formal array, it is the address of the array that is accessed.

C.3.7 Interaction with COBOL

The FORTRAN programmer may call COBOL programs as subprograms, and, conversely, the COBOL programmers may call FORTRAN-20 programs as subprograms.

In either of the foregoing cases, I/O operation must not be performed in the called subprogram.

C.3.7.1 Calling FORTRAN-20 Subprograms from COBOL Programs - COBOL programmers may write subprograms in FORTRAN to use the conveniences and facilities provided by this language. The COBOL verb ENTER is used to call FORTRAN-20 subroutines. The form of ENTER is as follows:

```
                                 ┌                                ┌                 ┐┐
                                 │      (identifier1      )       │   (identifier2) ││
ENTER FORTRAN program name       │USING <literal1         >       │ , <literal2    > ││
                                 │      (procedure name1  )       │   (procedure2  ) ││
                                 └                                └                 ┘┘
```

The USING clause of the foregoing forms names the data within the COBOL program that is to be passed to the called FORTRAN subprogram. The passed data must be in a form acceptable to FORTRAN-20.

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, the compiler creates an argument list that 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:

0-8	9-12	13-35
0	type	address

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 of the first word of the argument; the address can be indexed or indirect.

Following is a description of the types, their codes, how the codes appear in the argument list, and the locations specified by the addresses.

1. For 1-word COMPUTATIONAL items

CODE:	2
IN ARGUMENT LIST:	XWD 100, address
ADDRESS:	that of the argument itself

2. For 2-word COMPUTATIONAL items

CODE:	11
IN ARGUMENT LIST:	XWD 440, address
ADDRESS:	that of the high-order word of the argument

3. For COMPUTATIONAL-1 items

CODE:	4
IN ARGUMENT LIST:	XWD 200, address
ADDRESS:	that of the argument itself

4. 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 (via POPJ 17,) is to the statement after the call.

C.3.7.2 **Calling COBOL Subroutines from FORTRAN-20 Programs** - To call COBOL subroutines use the standard subroutine calling mechanism:

```
CALL COBOLS (args...)      subroutine call
X=COBOLS (args...)         function call
```

You must have compiled the COBOL subroutine using the COBOL compiler described in the DECsystem-20 COBOL Programmer's Reference Manual.

C.3.8 LINK Overlay Facilities

LINK provides several routines that are accessible directly from a FORTRAN-20 program. These routines are presented here briefly, together with the FORTRAN-20 specification of their parameters. In general, LINK performs these functions automatically. These routines are available only for your convenience. Full details of the use of the overlay facilities can be found in the LINK Reference Manual.

C.3.8.1 **Conventions** - The following terms are used to describe the parameters to LINK overlay routines.

File spec	A literal constant consisting of device: filename.ext [directory]
Name	A LINK name or number that is a literal constant or variable.
List of link names	A sequence of name items separated by commas.

The routines available are:

INIOVL	(File spec) Used to specify the overlay file to be found if the load time specification is to be overridden.
GETOVL	(List of link names) Used to change the overlay structure in memory.

RUNOVL	(Name) Loads the specified LINK and transfers to that LINK.
REMOVL	(List of link names) Removes the specified LINKs from memory.
LOGOVL	(File spec) Used to specify where the log file is to be written. If no arguments are given, the log file is closed.

For a full description of these routines, refer to the LINK Reference Manual.

APPENDIX D

FOROTS

This appendix describes the facilities that 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 primary 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

D.1 HARDWARE AND SOFTWARE REQUIREMENTS

FOROTS may interface with all DECsystem-20 peripheral devices. In addition to monitor or user program requirements, a minimum of 14 pages of user memory is needed to run FOROTS.

The software required with FOROTS is the TOPS-20 monitor, Version 1. Other software items that can be associated with FOROTS include:

1. The MACRO assembler
2. The LINK loader
3. The FORTRAN-20 compiler

D.2 FEATURES OF FOROTS

The following list briefly describes many specific features; more detailed information concerning the implementation of these features is given later in this appendix.

1. Your program may run in either batch or timesharing mode without requiring a program change. All differences between batch mode and timesharing mode operations are resolved by FOROTS.

2. Your programs may access both directory and non-directory devices in the same manner.

3. FOROTS helps provide complete data file compatibility between all system 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. You may change or specify object program device and file specifications 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 you 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 at the address of the point where the error occurred.

12. FOROTS provides a trap handling system for arithmetic functions, including default values and error reports.

13. You may mix ASCII and binary records in the same file, and both may be accessed in either sequential or random access mode.

14. FOROTS permits your 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 FORTRAN-20 compiler, you may also use FOROTS as an independent I/O system, as an I/O system for MACRO object programs, and for FORTRAN-20 object programs.

D.3 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 terminal or branches the program to a predesignated processing routine.

D.4 INPUT/OUTPUT FACILITIES

FOROTS uses monitor-buffered I/O during all modes of access except DUMP mode. Display devices are supported in dump mode; formatted text is handled in ASCII line mode; unformatted files are accessed as FORTRAN binary files. (Refer to the Monitor Calls User's Guide.)

The following paragraphs describe I/O data channel and access modes.

D.4.1 Input/Output Channels Used Internally by FOROTS

Fifteen software channels (1 through 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 (GETSEG UUO operations)

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.

D.4.2 File Access Modes

Data may be transferred between processor storage and peripheral devices in two major modes - sequential and random.

D.4.2.1 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 you 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.

You must specify transfer modes (other than SEQINOUT) by setting the ACCESS option of a FORTRAN-20 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)
```

D.4.2.2 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 (disk) that permits random addressing operations to files that have been set up for random access. Files for random access must contain a specified number of identically sized records that may be individually accessed by a record number.

You may accomplish random access transfers in either a read/write mode or a special read-only mode. You must specify random transfer modes 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)
```

D.5 ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS

The following paragraphs describe the types of data files that are acceptable to FOROTS.

D.5.1 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters. A vertical paper-motion character, such as, a Form Feed, a Vertical Tab, or a Line Feed, terminates each set. 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 physical 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.

D.5.2 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data that the executing program may reference with either a READ or WRITE statement. 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.

D.5.2.1 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 disk block boundary continues
END	003+	number of words in the preceding segment including LSCWs.

If the access you specify 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 you specify 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 LSCWs.

```
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 disk 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	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	
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        000277  000000 000007
000261  000000 000007        000300  000000 000007
000262  000000 000007        000301  000000 000007
000263  000000 000007        000302  000000 000007
000264  000000 000007        000303  000000 000007
000265  000000 000007        000304  000000 000007
000266  000000 000007        000305  000000 000007
000267  000000 000007        000306  000000 000007
000270  000000 000007        000307  000000 000007
000271  000000 000007        000310  000000 000007
000272  000000 000007        000311  000000 000007
000273  000000 000007        000312  000000 000007
000274  000000 000007        000313  000000 000007
000275  000000 000007        000314  003000 000147
000276  000000 000007
```

Image mode files contain no LSCWs. You cannot backspace this file.

```
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        000024  000000 000005
000001  000000 000005        000025  000000 000005
000002  000000 000005        000026  000000 000005
000003  000000 000005        000027  000000 000005
000004  000000 000005        000030  000000 000005
000005  000000 000005        000031  000000 000005
000006  000000 000005        000032  000000 000005
000007  000000 000005        000033  000000 000005
000010  000000 000005        000034  000000 000005
000011  000000 000005        000035  000000 000005
000012  000000 000005        000036  000000 000005
000013  000000 000005        000037  000000 000005
000014  000000 000005        000040  000000 000005
000015  000000 000005        000041  000000 000005
000016  000000 000005        000042  000000 000005
000017  000000 000005        000043  000000 000005
000020  000000 000005        000044  000000 000005
000021  000000 000005        000045  000000 000005
000022  000000 000005        000046  000000 000005
000023  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
```

D.5.3 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.

D.5.4 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 that 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.
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. All data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation.
Plotter	Six 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.

D.6 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 that 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 will provide automatic memory allocation, data conversion, I/O buffering, and device interface operations to the MACRO user.

D.6.1 FOROTS Entry Points

FOROTS provides 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
DBMS.	DBMS interface
DEC.	DECODE routine
DECHN.	De-allocate software channels
DECOR.	De-allocate dynamic memory blocks
ENC.	ENCODE routine
EXIT.	Terminate program exeuction
FIN.	Input/Output list termination routine
FIND.	Position to the next record (RANDOM ACCESS)
FORER.	Error processor
FUNCT.	Overlay interface
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

D.6.2 Calling Sequences

You must use the following general form for all calls made to FOROTS:

```
MOVEI  16,ARGBLK
PUSHJ  17,Entry Point
       (control is returned here)
```

where:

1. ARGBLK is the address of a specifically formatted argument block that contains information needed by FOROTS to accomplish the desired operation.

2. Entry Point is an entry point identifier (see list given in Paragraph D.6.1) that 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:

1. An error return to a specified statement number, i.e., READ or WRITE statement ERR=option,

2. An end-of-file return to a statement number, i.e., READ or WRITE statement END=option,

3. A fatal error that returns to the monitor or to a debug package.

Paragraphs D.6.3.1 through D.6.3.11 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 C. 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. In previous versions of FOROTS and FORTRAN-20, if the indirect bit was not set, the argument was immediate; if it was set to 1 (one), the argument was the address of the variable. The type field was always 0 (zero).

With Version 4 of FORTRAN-20 and Version 4 of FOROTS this convention has been changed. 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 C, Table C-1, 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)
```

D.6.3 MACRO Calls for FOROTS Functions

The following paragraphs describe the forms of the MACRO calls to FOROTS that are made by the FORTRAN-20 compiler. The calls described are identified according to the language statement that 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.)

u = a FORTRAN logical unit number

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	type	I	X	u
↓	7	I	X	c
↓	7	I	X	d
↓	type	I	X	f
↓	type	I	X	Format Size (in words)
Reserved	type	I	X	v

D.6.3.1 **I/O 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	type	I	X	u
↓	7	I	X	c
↓	7	I	X	d
↓	type	I	X	f
Reserved	type	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	type	I	X	u
↓	7	I	X	c
Reserved	7	I	X	d

D.6.3.2 **NAMELIST I/O, 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	type	I	X	u
↓	7	I	X	c
↓	7	I	X	d
Reserved	type	I	X	NAMELIST table address

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 you specify 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).

D.6.3.3 **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 (I1,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+(-L-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

D.6.3.4 **I/O 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
READ (u#R,END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.
```

and

```
WRITE (u#R,f,END=c, ERR=d) list
WRITE (u#R,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	type	I	X	u
↓	7	I	X	c
↓	7	I	X	d
↓	type	I	X	f
↓	type	I	X	format size (in words)
Reserved	2	I	X	address of Record Number

f and the format size in words are 0 if the I/O statement is unformatted.

D.6.3.5 **Calling Sequences for Statements That Use Default Devices** - The FORTRAN-20 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
↓	type	I	X	f
Reserved	type	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
-5				0
→ Reserved	2	I	X	u
↓	7	I	X	c
↓	7	I	X	d
↓	type	I	X	f
Reserved	type	I	X	format size (in words)

D.6.3.6 Statements to Posqition Magnetic Tape Units - The FORTRAN-20 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
-4				0
Reserved	type	I	X	u
	7	I	X	c
	7	I	X	d
Reserved	type	I	X	FOROTS code

D.6.3.7 List Directed Input/Output Statements - You may write any form of a sequential Input/Output statement 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

MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```
WRITE (u, *, 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

D.6.3.8 **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- or double-precision or complex 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	INCREMENT
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	144
0	0	0	0	1
0	2	0	0	A
4	0	0	0	0

More than one base address may appear in a SLIST as long as the increment is the same. The sequence

```
DIMENSION A(100), B(100)
WRITE (-,-) (A(I),B(I),I=100) ! only when the /OPT
                                switch is used
```

develops a SLIST argument of the form:

0-8	9-12	13	14-17	18-35
0				
2	0	0	0	144
0	0	0	0	1
0	2	0	0	A
0	2	0	0	B
4	0	0	0	0

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	type	I	X	No. Elements to transfer increment 1
	type	I	X	Base ADDR 1 increment 2
	type	I	X	Base ADDR 2 increment N
	type	I	X	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	0	0	0	144
0	0	0	0	1
0	2	0	0	IB
0	0	0	0	12
0	2	0	0	IC
4	0	0	0	0

The increment may be zero. This could be produced by the sequence

```
DIMENSION A(100)
WRITE(-,-)(K,I=100)   !only when the /OPT switch is used
```

The zero may not appear as an immediate constant in the argument block. The ELIST for the previous example would be

0-8	9-12	13	14-17	18-35
3	0	0	0	144
0	2	0	0	Pointer to a word containing a zero
0	type	0	0	K
4	0	0	0	0

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 rout^ine 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:

```
PUSHJ 17, FIN.
```

D.6.3.9 **OPEN and CLOSE Statements, Calling Sequences** - The form and calling sequences for the OPEN and CLOSE 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 number of words in block not including this one.				0
0	2	I	X	u
0	7	I	X	c
0	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 through 8) contains a 2-digit numeric that defines the argument name; the H field (bits 18 through 35) contains an address which points to the value of the argument.

The numeric codes that may appear in the G field and the argument that each identifies are:

G Field	Open Argument	G Field	Open Argument
01	DIALOG	12	MODE
02	ACCESS	13	FILE SIZE
03	DEVICE	14	RECORD SIZE
04	BUFFER COUNT	15	DISPOSE
05	BLOCK SIZE	16	VERSION
06	FILENAME	22	ASSOCIATE VARIABLE
07	PROTECTION	23	PARITY
10	DIRECTORY	24	DENSITY

D.6.3.10 **Memory Allocation Routines** - The memory management module is called to allocate or de-allocate memory blocks. There are two entry points, ALCOR. and DECOR., that control memory allocation and de-allocation.

Use the ALCOR. entry to allocate the number of words specified in the argument block variable. Upon return, AC 0 will contain either the address of the allocated memory block or a -1 value, which indicates that memory is not available. The calling sequence for ALCOR. call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, ALCOR.
```

where ARGBLK is

	0-8	9-12	13	14-17	18-35
	-1				0
→	Reserved	type	I	X	Address of Number of Words

Use the DECOR. entry to de-allocate a previously allocated block of memory; the argument variable must be loaded with the address of the memory block to be returned. Upon return AC 0 is set to 0.

If the number of desired words is N, ALCOR. actually removes N+1 words from free storage. The pointer returned points to the second word (word 1 as opposed to word 0) removed from free storage. The 0 word contains the negative value of N in its left half. This word is used by FOROTS to maintain linked lists of allocated (using ALCOR.) and free storage.

The calling sequence for a DECOR. call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, DECOR.
```

where ARGBLK is

0-8	9-12	13	14-17	18-35
-1				0
→ Reserved	type	I	X	Pointer to word containing address of block to be returned

D.6.3.11 **Software Channel Allocation and De-allocation Routines** - You may allocate software channels in MACRO programs via calls to the ALCHN. routine and de-allocate them by calls to the DECHN. routine. Values are returned in AC 0.

Use the ALCHN. entry to allocate a particular channel or the next available channel. The channel to be allocated is passed to ALCHN. in the argument block variable. Zero is passed in the argument block variable to allocate the next available channel. Allowed channels are 1 through 17 (octal). If the channel requested is not available, or all channels are in use, ALCHN. returns with a -1 in AC 0. In normal returns, AC 0 contains the assigned number.

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

Use the DECHN. entry 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

D.7 FUNCTIONS TO FACILITATE OVERLAYS

FOROTS provides a subroutine (FUNCT.) to serve as an interface with the LINK overlay handler. This subroutine consists of a group of functions that allow the overlay handler to perform I/O, memory management, and error message handling. These functions have only one entry point, FUNCT., and they are called by the sequence

```
MOVEI 16, ARGBLK
PUSHJ 17, FUNCT.
```

The general form of the ARGBLK is

	0-17	18-35
	Negative of the number of words in block	0
ARGBLK→	type	function number
	type	error code
	type	status
	type	argument 1
	type	argument 2
	type	argument 3
	.	.
	.	.
	type	argument n

where

type	= the FORTRAN argument type (see Appendix C)
function number	= the number of one of the required functions
error code	= the 3-letter mnemonic output by the object time system after ?, %, or [. (See Table D-1.)
status	= undefined on the call and set on the return with one of the values below.

-1	Function not implemented
0	Successful return
1....n	Specific error message

Table D-1
Function Numbers and Function Codes

Function Number	Function Mnemonic	Function Description
0	ILL	Illegal function
1	GAD	Allocates memory from a specific address
2	COR	Allocates memory from available core
3	RAD	De-allocates memory
4	GCH	Gets or assigns an I/O channel
5	RCH	Releases an I/O channel
6	GOT	Allocates memory from FOROTS
7	ROT	De-allocates memory from FOROTS
8	RNT	Returns the initial runtime from FOROTS
9	IFS	Returns initial runtime file spec. from FOROTS
10	CBC	Cuts back memory if possible

FUNCTION 0 (ILL) - This function is illegal. The argument block is ignored, and the function always returns a status of -1.

FUNCTION 1 (GAD) - This function allocates memory from a specific address. The arguments are:

arg 1 address at which to begin core allocation
arg 2 number of words of memory to allocate

The return statuses are:

0 core allocated (arg 1 and 2 unchanged)
1 not enough memory available in system (arg 1 and arg 2 unchanged)
2 cannot allocate memory at specified address (arg 1 and arg 2 unchanged)
3 illegal arguments (i.e., address + size is greater than 256K) (arg 1 and arg 2 unchanged)

FUNCTION 2 (COR) - This function allocates memory from any address. The arguments are:

arg 1 undefined
arg 2 size of core to allocate

The returned statuses are:

0 core allocated (arg 2 unchanged, arg 1 beginning address of the allocated memory)
1 not enough memory available in system (arg 2 unchanged)
3 illegal argument (i.e., size is greater than 256K)

FUNCTION 3 (RAD) - This function de-allocates memory at the specified address. The arguments are:

arg 1 address of core to be de-allocated
arg 2 number of words to be de-allocated

The returned statuses are:

0 memory de-allocated
1 memory cannot be de-allocated
3 illegal argument (i.e., both the address and the size are greater than 256K)

FUNCTION 4 (GCH) - This function assigns an I/O channel. The argument is:

arg 1   undefined

The returned statuses are:

0 I/O channel assigned (arg 1 channel number)
1 no I/O channels available

FUNCTION 5 (RCH) - This function releases an I/O channel. The argument is:

arg 1   I/O channel number to be released

The returned statuses are:

0 channel released
1 invalid channel number

FUNCTION 6 (GOT) - This function gets memory from the object time system list. The arguments are:

arg 1   address at which to allocate memory
arg 2   number of words of memory to allocate

The returned statuses are:

0 memory allocated (arg 1 and arg 2 unchanged)
1 not enough memory available in system (arg 1 and arg 2 unchanged)
2 cannot allocate memory at specified address (arg 1 and arg 2 unchanged)
3 illegal argument(s)

This function differs from function 1 in that if the object time system has two free memory lists, then function 1 is used to allocate space for links, and this function is used to allocate space for I/O buffers. Function 1 uses the free memory list for LINK, and function 6 uses the list for the object time system.

FUNCTION 7 (ROT) - This function returns memory to the object time system. The arguments are:

arg 1   address of memory to be de-allocated and returned
arg 2   size of memory to be de-allocated and returned

The returned statuses are:

0 memory de-allocated
1 memory cannot be de-allocated
3 illegal argument

FUNCTION 8 (RNT) - This function returns the initial runtime from the object time system. The argument is:

arg 1   undefined

The returned status is:

0 always (arg 1 - runtime from the object time system)

This function is used only if the user desires a log file.

FUNCTION 9(IFS) - This function returns the initial runtime file specification from the object time system. The specification is obtained from accumulators 0, 7, and 11 after the initial RUN command. The arguments are:

arg 1 undefined
arg 2 undefined
arg 3 undefined

The returned status is:

0 always (arg 1 - device from accumulator 11, arg 2 - filename from accumulator 0, and arg 3 - directory from accumulator 7)

This function tells the overlay handler which file to read after the initial RUN command.

FUNCTION 10 (CBC) - This function cuts back memory if possible and is used to reduce the size of the user job. There are no arguments.

The returned status is:

0 always

D.8 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

You make FORTRAN logical and physical device assignments at run time, or standard system assignments are made according to a FOROTS Device Table, i.e., DEVTB. Table D-2 shows the standard assignments contained by the Device Table.

Table D-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
	-2	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
	06 through 15 not valid	
MTA0	16	Magnetic Tape
MTA1	17	Magnetic Tape
MTA2	18	Magnetic Tape
FORTR	19	Assignable Device
DSK	20	DISK
DSK	21	DISK
DSK	22	DISK
DSK	23	DISK
DSK	24	DISK
DEV1	25	Assignable Devices
DEV2	26	
DEV3	27	
DEV4	28	
DEV5	29	
.	.	
.	.	
.	.	
DEV39	63	

APPENDIX E

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 you to examine and modify the data and FORMAT statements in your program, set breakpoints at any executable statement or routine, trace your program statement-by-statement, and make use of many other debugging techniques described in this appendix.

Table E-1 lists all the commands available to the user of FORDDT.

Table E-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. See Table B-2.)
DOUBLE	Defines dimensions of double-precision arrays for FORDDT references. (Unnecessary if /DEBUG: DIMENSIONS was used. See Table B-2.)

Table E-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.
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.

E.1 INPUT FORMAT

FORDDT commands are made up of alphabetic FORTRAN-like identifiers and need consist of only those characters required to make the command unique. If you wish to specify parameters, a space or tab is required following the command name. FORDDT expects a parameter if a delimiter (i.e., space or tab) is found. Comments may be appended to command lines by preceding the comment with an !.

E.1.1 Variables and Arrays

FORDDT allows you to access and modify the data locations in your program by using standard FORTRAN symbolic names. Variables are specified simply by name. Array elements are specified in the following format:

```
name (S1,...,Sn)
```

where

```
name          = a FORTRAN variable or array name
(S1,...,Sn)   = the subscripts of the particular array.
```

You may reference an entire array simply by its unsubscripted name; you may specify a range of array elements 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)
```

E.1.2 Numeric Conventions

FORDDT accepts optionally signed numeric data in the standard FORTRAN-20 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.

E.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.

E.2 NEW USER TUTORIAL

The new FORDDT user can rely on the commands described below as a basis for debugging FORTRAN programs. These commands are easy to understand and apply.

E.2.1 Basic Commands

The easiest method of loading and starting FORDDT is:

```
@DEBUG filename.ext/FORTRAN/DEBUG
```

FORDDT will respond with

```
ENTERING FORDDT
>>
```

Just as an asterisk (*) signifies FORTRAN-20'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 your 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. You need type only the first character of the mode to identify it to FORDDT. The modes are:

Mode	Meaning
A	ASCII (left-justified)
C	COMPLEX
D	DOUBLE-PRECISION
F	FLOATING-POINT
I	INTEGER
O	OCTAL
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 you to display the contents of one or more data locations. They are displayed on your 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 you 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 the elements to be set to the given value (see special case for ASCII in Section E.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

You need type only the first character of a data mode 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 you to set a breakpoint at any label, line number, or subroutine entry in your program. You may set up to ten pauses 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 you include a value 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 your terminal the name of the currently open program unit and any currently active pause settings. The command format is:

WHAT

E.3 FORDDT AND THE FORTRAN-20/DEBUG SWITCH

Most facilities of FORDDT are available without the FORTRAN-20 /DEBUG features; however, if you do not use the /DEBUG switch 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-20 to compile extra information for FORDDT. (See Appendix B, 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 you wish 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, you will be able to display and modify 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 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. You will then be able to examine DO loops by using FORDDT. If you modify a DO loop index using FORDDT, it 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.

E.4 LOADING AND STARTING FORDDT

1. The simplest method of loading and starting FORDDT is with the following command string:

   ```
   @DEBUG filename.ext/FORTRAN/DEBUG
   ```

FORDDT responds with

```
ENTERING FORDDT
>>
```

The angle brackets indicate that FORDDT is ready to receive a command, just as an asterisk (*) signifies FORTRAN-20's readiness.

The DEBUG command to the monitor will also load DDT (standard system debugging program). DDT can be used or ignored.

2. You may wish to load your compiled program and FORDDT directly with the LINK loader. (Loading with LINK was accomplished implicitly in the previous command string.) The command sequence is as follows:

```
@LINK
*filename.ext /DEB/G                              (loads DDT)
*filename.ext /DEB:  FORDDT  /G                   (loads FORDDT)
                     FORTRA

*filename.ext /DEB:(DDT,  FORDDT  )/G             loads both DDT
                          FORTRA                   and FORDDT
```

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.

E.5 SCOPE OF NAME AND LABEL REFERENCES

Each program unit has its own symbol table. When you initially enter FORDDT, you automatically open 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 you have 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 you enter 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.5.)

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.

E.6 FORDDT COMMANDS

This section gives a detailed description of all commands in FORDDT. The commands are given in alphabetical order.

ACCEPT Allows you 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 long.

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, FORDDT 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 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 you 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
```

where S is the name of the array specified.

For example:

```
DIMENSION ALPHA(7,5/6,10)
```

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 [d1/d2,...](S1,...)
```

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 you to continue your program from a point other than the one at which it last paused. The GOTO allows you 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 ^C^C REENTER sequence. (See Section E.7.)

The command format is:

```
GOTO n
```

GROUP Sets up a string of text for input to a TYPE command. You 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 you wish 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 you wish 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 you wish 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 you to cause FORDDT to trace source lines, statement labels, and entry point names during execution of your program. This command will only provide trace facilities if the program was compiled with the FORTRAN-20 /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 you 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 you will be requested upon entering FORDDT.

PAUSE — Allows you to place a pause request at a statement number, source line number, or subroutine entry point. Up to ten 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 constant or variable or array element
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 control C REENTER sequence, a START command, or a control C START sequence.

REMOVE — Removes the previously requested pauses. The command format is:

REMOVE P

For example,

REMOVE #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 your pause requests, and, in this case, no abbreviation of REMOVE is allowed. This prevents the unintentional removal of pauses.

START — Starts your 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 your 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 you do not specify a format, the default is floating-point form.

2. You can specify a format via the MODE command described in this appendix.

3. You 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 /IK,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
```

E.7 ENVIRONMENT CONTROL

If a program enters an indefinite loop, you can recover by typing a ^C^C REENTER sequence. This action will cause FORDDT to simulate a pause at the point of reentry and allow you to control your 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 ensure that any of these commands following a ^C^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, you 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.

E.8 FORTRAN-20/OPTIMIZE SWITCH

You 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 ACs. For this reason, attempts to examine or modify variables in optimized programs will not work. Also, since the optimizer moves statements around in your program, attempts to trace program flow will lead to great confusion.

E.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-20 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-20 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-20.

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. Refer to the documentation for correct format.

IAT ILLEGAL ARGUMENT TYPE = [number]

An unrecognized subprogram argument type was detected. Submit an SPR if this message occurs.

ICC COMPARE TWO CONSTANTS IS NOT ALLOWED

Conditional test involves two constants.

IER E (number)

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 6-character alphanumeric 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. The maximum limit is 10.

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 fewer 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 your 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 "text" 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 ensure that the program is properly initialized.

POV PROGRAM OVERLAYED

The symbol table is different from the last time FORDDT had control.

SFA SUPERSEDES F10 ARRAY

The 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 the currently defined program space.

APPENDIX F

COMPILER MESSAGES

FORTRAN-20 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.

ATL ARRAY [name] TOO LARGE

The total amount of memory necessary to accommodate this array is greater than 512P.

AWN ARRAY REFERENCE [name] HAS WRONG NUMBER OF SUBSCRIPTS

The array was defined to have more or fewer 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 syntactic analysis begins. During this classification the entire portion of the required statement 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-20 has classified a statement, there is no explicit restriction on its length.

CER COMPILER ERROR IN ROUTINE [name]

Submit an SPR for any occurrence of this message.

CFF CANNOT FIND FILE

The file referenced in an INCLUDE statment was not found.

CPE CHECKSUM OR PARITY ERROR IN [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-20.

DNL IMPLIED DO SPECIFICATION WITHOUT ASSOCIATED LIST OF VARIABLES

DPR DUMMY PARAMETER [name] REFERENCED BEFORE DEFINITION

DSF ARGUMENT [name] IS SAME AS FUNCTION NAME

DTI THE DIMENSIONS OF [arrayname] MUST BE OF THE TYPE INTEGER

DVE CANNOT USE DUMMY VARIABLE 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 EITHER [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 [symbol]

HDE HARDWARE DEVICE ERROR ON [source/listing/object] DEVICE [[device]]

IAC ILLEGAL ASCII CHARACTER [character] IN SOURCE

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

You are attemping 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 A 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 THAN 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 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.

NIO NAMELIST DIRECTED I/O WITH I/O LIST

NGS CANNOT GET SEGMENT [name] - ERROR CODE: [number]

Refer to the Monitor Calls User's Guide for full description of codes.

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

PD6 FORTRAN WILL NOT RUN ON A PDP-6

PIC THE DO PARAMETERS OF [index name] MUST BE INTEGER CONSTANTS

PRF PROTECTION FAILURE [filename]

PTL PROGRAM TOO LARGE

The program takes up more than 512P

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 LABELED

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.

UCE USER CORE EXCEEDED

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

VSO VARIABLE DIMENSION ALLOWED IN SUBPROGRAMS ONLY

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 expessions at compile time.

DIM POSSIBLE DO INDEX MODIFIED INSIDE LOOP

A program that 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 OPT - PROGRAM IS DISCONNECTED - OPTIMIZATION DISCONTINUED

Submit an SPR if this message occurs.

DXB DATA STATEMENT EXCEEDS BOUNDS OF ARRAY [name]

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 that cannot be reached during execution.

ICS ILLEGAL CHARACTER IN LINE SEQ#

IDN OPT - ILLEGAL DO NESTING - OPTIMIZATION DISCONTINUED

A GO TO within a DO loop goes to the ending statement of an inner, nested DO loop. The line number printed out with the warning message is that of the OUTER DO.

```
DO
.
.
.
GO TO
.
.
```

```
        .
        DO
        .
        .
        .
        CONTINUE
        .
        .
        .
        CONTINUE
```

IFL — OPT - INFINITE LOOP. OPTIMIZATION DISCONTINUED

LID — IDENTIFIER [name] MORE THAN SIX 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

VAI — [name] ALREADY INITIALIZED

VND — FUNCTION RETURN VALUE IS NEVER DEFINED

VNI — OPT - 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 - WARNING GIVEN IN PHASE 1. OPTIMIZED CODE MAY NOT BE CORRECT

One or more of the messages issued prior to this message resulted from situations that 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 that is accepted only for compatibility.

Internal Compiler errors

An internal compiler error is either an attempt by the compiler or the monitor to document an error inside the FORTRAN compiler. An occurrence of an internal compiler error signifies that something is wrong with the FORTRAN-20 compiler.

Monitor-detected internal errors are of the form

[message] AT LOCATION [address] IN PHASE [segment]

WHILE PROCESSING STATEMENT [line-number]

where [message] can be one of

ILLEGAL MEMORY REFERENCE

STACK EXHAUSTED

MEMORY PROTECTION VIOLATION

Compiler-detected errors are of the form

? INTERNAL COMPILER ERROR PROCESSING STATEMENT NUMBER [line-number]

? CALL TO [routine-name] FROM [address]

Submit an SPR if you received an internal compiler error.

APPENDIX G

FOROTS ERROR MESSAGES

Errors detected at run-time by FOROTS fall into the following categories:

1. system errors (SYS) - errors internal to FOROTS
2. open errors (OPN) - I/O errors that occur during file OPEN and CLOSE
3. arithmetic fault errors (APR) - errors in numeric calculations
4. library errors (LIB) - errors generated by FORLIB library routines
5. data errors (DAT) - errors in data conversion on I/O
6. device errors (DEV) - I/O hardware errors

APR and LIB errors are usually reported as warnings and the program continues. The number of APR and LIB errors listed on the user's terminal can be changed by the FORTRAN Library Subroutine ERRSET. See Table 15-3 for details. The I/O errors (SYS, OPN, DAT, and DEV) either cause messages to be printed on the terminal or can be trapped by an error exit argument (ERR=statement label) on OPEN, READ, WRITE, and CLOSE.

Table G-1 gives the text of the messages which can be printed for SYS, OPN, DAT, and DEV errors. The included footnotes give additional information. Table G-2 gives the text of the messages which can be printed for APR and LIB errors.

The FORTRAN Library Subroutine ERRSNS allows you to find out which I/O error occurred. When called, ERRSNS returns one or two integer values that describe the status of the last I/O operation performed by FOROTS. (The second integer value is optional.)

```
CALL ERRSNS (I,J)
```

calls this subroutine. J is the second, optional integer value.

Table G-1
FOROTS I/O Error Messages and ERRSNS Returned Values

First Value	Second Value	Explanation
0		No error detected
	0	Satisfactory completion (no error detected)
	101	Normal end of job (1)
1		Invalid error call
	243	Unidentified entry in FORERR (3)
	246	Unidentified entry in FORERR (3)
23		Backspace error
	312	BACKSPACE illegal for device (9)
24		End-of-file during READ
	308	Attempt to READ beyond valid input (8)
25		Invalid record number
	302	LSCW illegal in binary record or reading ASCII; or attempt to read unwritten ASCII RANDOM ACCESS record or unwritten or destroyed record number
26		Direct access not specified
	311	Cannot RANDOM ACCESS a SEQUENIAL file
28		CLOSE error
	252	DTA directory is full (2) or protection error
	254	Rename file already exists (2)
	262	No room or quota exceeded (2)
	268	Cannot delete or rename a non-empty directory (2)
29		No such file
	250	File was not found
30		OPEN failure
	237	DUMP mode RANDOM or APPEND access not implemented; try IMAGE MODE
	238	DIALOG file cannot be opened (3)
	240	Record length missing for RANDOM ACCESS
	242	Too many devices open: fifteen maximum
	245	Device not available (2)
	248	Illegal ACCESS for device (2)
	249	Illegal MODE or MODE switch (2)
	251	No directory for project, programmer number (2)
	253	File was being modified (2)

1. Not currently implemented.

2. OPEN errors 251 through 276 map directly onto error numbers returned by the OPEN UUO; see the Monitor Calls Manual.

3. Error cannot currently occur.

8. Occurs when simulating mag tape output; SKIP RECORD and SKIP FILE are illegal. Also occurs when a non-existent file is opened in MODE=SEQINOUT and the first operation on that file is a READ.

9. Occurs if OPEN output with BACKSPACE is not a mag tape or disk.

Table G-1 (Cont.)
FOROTS I/O Error Messages and ERRSNS Returned Values

First Value	Second Value	Explanation
	255	Illegal sequence of Monitor Calls (11)
	256	Bad UFD or bad RIB (2)
	259	Device not available (2)
	265	Partial allocation only (2)
	266	Block not free on allocation (2)
	267	Cannot supersede an existing directory (2)
	269	SFD not found (2)
	270	Search list empty (2)
	271	SFD nested too deeply (2)
	272	No CREATE flag for specified UFD (2)
	274	File cannot be updated (2)
	277	LOOKUP ENTER or RENAME error (2)
31		Mixed access modes
	315	Cannot do SEQUENTIAL ACCESS on a RANDOM file
32		Invalid logical unit number
	239	Illegal FORTRAN unit number (2)
39		Error during READ
	310	REREAD before first READ is illegal (1)
42		Device handler not resident
	244	No such device (2)
	260	No such device (2)
45		OPEN statement keyword error
	241	Switch error during DIALOG or OPEN statement scan (2)
47		Write on read-only file
	263	Write-lock error (2)
59		List-directed I/O syntax error
	313	Illegal delimiter in LIST DIRECTED input
62		Syntax error in FORMAT
	301	Illegal character in FORMAT statement (4)
	306	I/O list without data conversion in FORMAT
	314	Missing width field for A or R on input
63		Output conversion error
	305	Optional * fill: unidentified entry in FORERR (7)

1. Not currently implemented.

2. OPEN errors 251 through 276 map directly onto error numbers returned by the OPEN UUO; see the Monitor Calls Manual.

4. In runtime FORMAT.

7. * fill controlled by compile-time variable ASTFIL.

11. Can occur on OPEN (MODE= 'APPEND') when file is found in LIB: or on [1,4] when device specified was SYS: and /NEW was in your search list.

Table G-1 (Cont.)
FOROTS I/O Error Messages and ERRSNS Returned Values

First Value	Second Value	Explanation
64		Input conversion error
	303	Checksum error reading binary records (5)
	307	Illegal character in data
67		Record too small for I/O list
	304	I/O list greater than record size (6)
81		Invalid argument
	102	Argument block not in correct format
	261	Argument block not in correct format (2)
699		Unclassifiable error on OPEN
	247	FOROTS system error (2,3)
	257	FOROTS system error (2)
	258	FOROTS system error (2)
	264	Not enough monitor table space (2)
	273	FOROTS system error (2)
	275	FOROTS system error (2)
	276	FOROTS system error (2)
799		Unclassifiable data error
	309	Variable cannot be found in NAMELIST block
899		Unclassifiable device errors
	400	Write protected
	401	Device error
	402	Parity error
	403	Block too large, quota exceeded, or file structure full. Nonexistent CDR reader. Spooled CDR file does not exist.
	404	End-of-file (10)
	407	End-of-tape
999		Unclassified system error
	100	FOROTS system error
	103	Monitor not build to support FOROTS
	104	Fatal error
	105	User program has requested more code than is available
	106	Run time memory management error

2. OPEN errors 251 through 276 map directly onto error numbers returned by the OPEN UUO; see the Monitor Calls Manual.

3. Error cannot currently occur.

5. Checksumming controlled by compile-time variable CHKSUM.

6. Occurs when a type 2 LSCW is found in a FORSE binary record.

10. Trappable if there is no END= clause.

Table G-2
FOROTS Arithmetic and Library Error Messages

APR	LIB
Integer Overflow	Attempt to take DLOG of Negative Arg.
Integer Divide Check	Attempt to take DSQRT of Negative Arg.
Illegal APR Trap	ACOS of Arg. > 1.0 in Magnitude
Floating Divide Check	ASIN of Arg. > 1.0 in Magnitude
Floating Underflow	Attempt to take SQRT of Negative Arg.
	Attempt to take LOG of Negative Arg.

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:

   SLITE(i)
   SLITET(i,j)
   SSWTCH(i,j)

5. DDT command to FORDDT.

INDEX

DECsystem-20
FORTRAN Reference Manual
AA-4158B-TM

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an 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 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 ____________

Please cut along this line.

-- **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 Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital