PART 7

THE DOS/BATCH FORTRAN COMPILER AND OBJECT TIME SYSTEM

PART 7

CHAPTER 1

INTRODUCTION TO FORTRAN

FORTRAN (FORmula TRANslation) is a problem-oriented language designed to permit scientists and engineers to express mathematical operations in a form with which they are familiar. The term FORTRAN is used interchangeably to designate both the FORTRAN language and the FORTRAN Compiler or translator. A program written in the FORTRAN language is called a FORTRAN source program.

A FORTRAN source program is composed of commands describing the functions performed and computational elements expressed in a notation similar to that of mathematics. The source program is compiled by the FORTRAN Compiler into code which is then automatically assembled into a binary object module. The Linker program links the resultant relocatable module with any other required module or modules to create an executable program image called a load module. Execution is started by using the appropriate commands as described in Chapter 7-9 of this manual.

There is no one-to-one correspondence between a FORTRAN statement and a machine-language instruction. Some FORTRAN statements generate many machine instructions while others yield none.

1.1 DOS/BATCH FORTRAN

DOS/BATCH FORTRAN conforms to the specifications for American National Standard FORTRAN. It is also highly compatible with IBM 113Ø FORTRAN. The following additions to American National Standard FORTRAN are included in DOS/BATCH FORTRAN:

a. An array subscript may consist of any valid arithmetic expression. If the result of the expression is not an integer, it is truncated and converted to integer format before being used as a subscript.

b. Hexadecimal, octal and Radix-5Ø constants are allowed within DATA statements.

c. Hollerith constants may be delimited by single quote (apostrophe) characters.

d. General mixed-mode expressions are allowed for all data types, including complex.

e. Assigned GOTO statements may contain optional label lists.

f. The following statements have been added to handle random access I/O:

```
DEFINE FILE
FIND
READ (a'b)
WRITE (a'b)
```

g. The IMPLICIT statement is provided to allow redefinition of the rules for data type classification of variables by default. (In the absence of any data type specification, a variable name beginning with one of the letters I through N indicates an integer variable; A through H and O through Z indicate real variables.)

h. The data types

```
LOGICAL*1
INTEGER*2
REAL*4
REAL*8
```

are provided.

j. The I/O conditions END=n and ERR=n can be specified on any READ or WRITE statement to transfer control upon detection of an end-of-file or error condition.

k. Default values for field width and number of characters to the right of the decimal point are provided on numeric and logical format conversions.

l. Variable format expressions are provided.

m. A simplified type of free form formatted input (termed "short field termination") is provided.

1.2 SOFTWARE AND HARDWARE ENVIRONMENTS

The FORTRAN Compiler produces code suitable for assembly by MACRO. The user can compile and assemble his FORTRAN program in one step (which is normally done), or he can compile the program into assembly language for later assembly (indicated by means of a switch option to the compiler). FORTRAN programs can call assembly language subroutines. The FORTRAN Library routines can be called by either FORTRAN or assembly language code.

DOS/BATCH FORTRAN programs can be compiled and run on any hardware configuration that supports the DOS/BATCH operating system, and that has a minimum of 16K of memory. DOS/BATCH FORTRAN supports all standard hardware options supported by the operating system.

1.3 HOW TO USE THIS PART

This Part is primarily a reference document describing the DOS/BATCH FORTRAN language. It is not intended to teach the reader how to program in FORTRAN. The bulk of this Part describes details of the language statements and descriptions of the file handling, library, internal structures, and Object Time System. Later chapters provide concise summaries of material in tabular form.

Although designed as a reference tool, this Part contains a number of programming examples to show the use of statements in a context. Internal details supplied will aid the more advanced FORTRAN programmer.

1.4 FORTRAN STATEMENT STRUCTURE

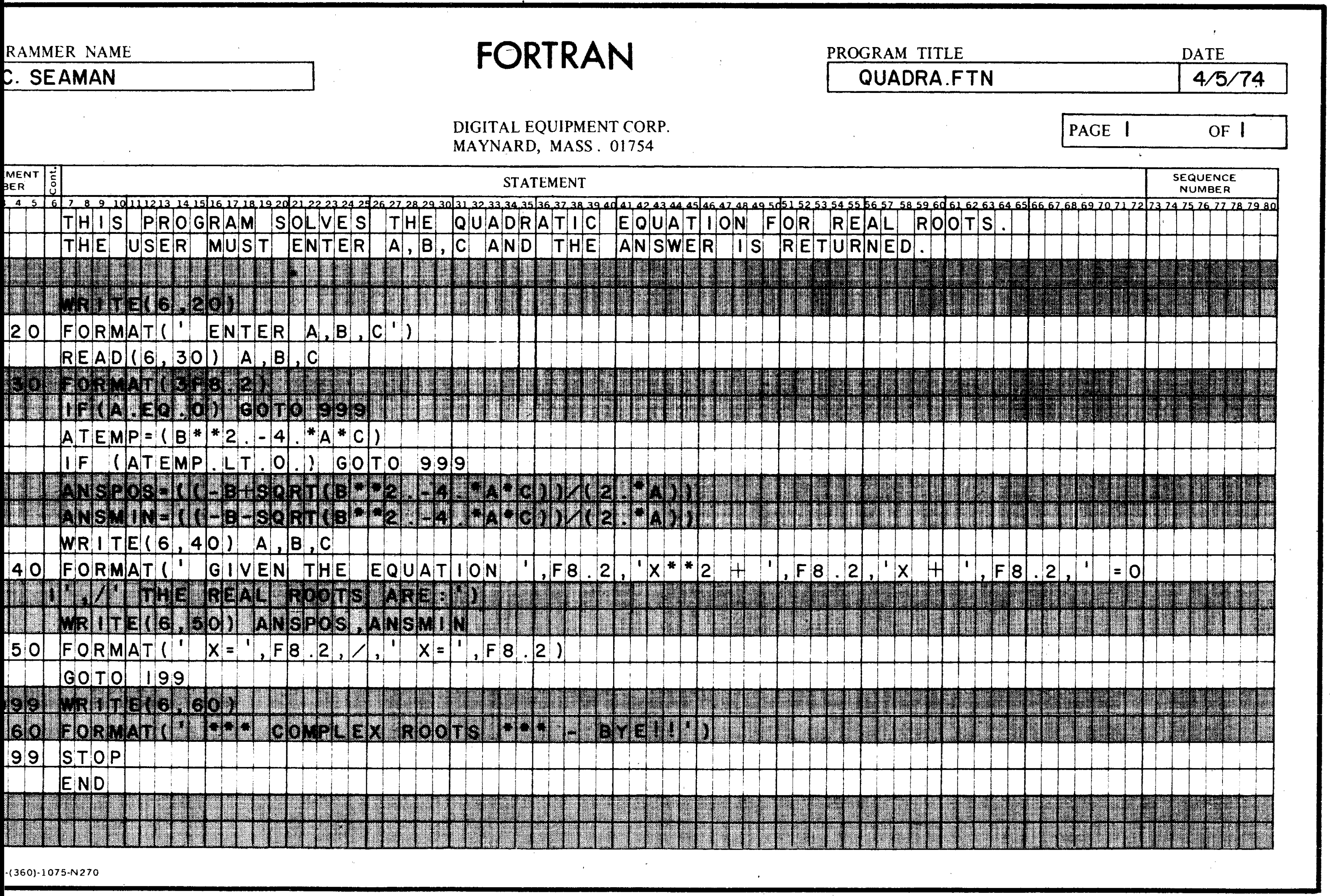
A statement is the basic functional unit of the FORTRAN language. An executable statement performs a calculation, does I/O, or directs control of the program; a nonexecutable statement provides the Compiler with information regarding variable structure, array allocation, storage requirements, etc.

All categories of FORTRAN statements are coded in a standard format for ease of program preparation. Figure 7-1 shows a FORTRAN coding form (there are many types available) with four miscellaneous lines written on it. FORTRAN coding forms lend themselves to paper tape input as well as to punched card input preparation. The form is divided into four major sections:

a. Statement number field - columns 1-5.
b. Line continuation field - column 6.
c. Statement field - column 7-72.
d. Sequence field (ignored by the FORTRAN Compiler) - columns 73-80.

1.4.1 Formatting a FORTRAN Line

Punched card input for the DOS/BATCH FORTRAN Compiler is formatted as shown on the coding form in Figure 7-1. This is the standard format for FORTRAN punched cards on most computer systems.



RAMMER NAME C. SEAMAN

FORTRAN

PROGRAM TITLE QUADRA.FTN

DATE 4/5/74

DIGITAL EQUIPMENT CORP.
MAYNARD, MASS. 01754

PAGE 1 OF 1

MENT BER | Cont. | STATEMENT | SEQUENCE NUMBER

```
      THIS PROGRAM SOLVES THE QUADRATIC EQUATION FOR REAL ROOTS.
      THE USER MUST ENTER A,B,C AND THE ANSWER IS RETURNED.

      WRITE(6,20)
   20 FORMAT(' ENTER A,B,C')
      READ(6,30) A,B,C
   30 FORMAT(3F8.2)
      IF(A.EQ.0) GOTO 999
      ATEMP=(B**2.-4.*A*C)
      IF (ATEMP.LT.0.) GOTO 999
      ANSPOS=((-B+SQRT(B**2.-4.*A*C))/(2.*A))
      ANSMIN=((-B-SQRT(B**2.-4.*A*C))/(2.*A))
      WRITE(6,40) A,B,C
   40 FORMAT(' GIVEN THE EQUATION ',F8.2,'X**2 + ',F8.2,'X + ',F8.2,' =0
     1',/' THE REAL ROOTS ARE:')
      WRITE(6,50) ANSPOS,ANSMIN
   50 FORMAT(' X=',F8.2,/,' X=',F8.2)
      GOTO 199
   99 WRITE(6,60)
   60 FORMAT(' *** COMPLEX ROOTS *** - BYE!!')
   99 STOP
      END
```

-(360)-1075-N270

Figure 7-1

FORTRAN Coding Form

When programs are prepared on-line with the Editor, statements can be typed in the same manner as described for cards or they can use the TAB character as follows:

a. A TAB causes the internal character count to be set to 6 or 7 depending on whether or not the character following the TAB is numeric. If the character is not numeric, it is assumed to be the start of a statement proper. If the character following the tab is numeric, the line is assumed to be a continuation line.

b. A 1- to 5-digit statement label may precede the TAB character.

In order to obtain the formatting shown on the coding form in Figure 7-1, the user would type the following to EDIT:

```
123→|X=12.+3.          or         123ΔΔΔX=12.+3

→|A=B+C                or         ΔΔΔΔΔΔA+B+C
→|1A1'ABCƀ'            or         ΔΔΔΔΔ1A1'ABC ' (continuation line)
 →|COMMENT LINE        or         CΔΔΔΔΔCOMMENT LINE
```

where:

→| - indicates typing of the TAB character (CTRL/TAB)
Δ - indicates typing of the SPACE bar.

As specified in Section 7-2.1, the TAB character is not included in the legal FORTRAN character set; therefore, the TAB character can be used only at the beginning of a line to indicate spacing to the continuation or statement field. FORTRAN ignores spaces within each field (except within Hollerith constants); spaces may therefore be inserted at the user's convenience to increase program legibility.

1.4.2 Statement Number Field

A statement number consists of one to five digits in columns 1 to 5. Leading zeros and all blanks in this field are ignored. Any statement to which reference is made by another statement must have a statement number. A label consisting only of one or more zero characters is not allowed.

Statement numbers must be unique. If two statements within a program unit have the same statement number, the error is noted at assembly time. There is no limit to the value of the statement number.

1.4.3 Continuation Lines

Any FORTRAN statement may be continued onto more than one physical source line. A nonzero or nonblank character is placed in column 6 of all continuation lines (or cards). When using an editor, the TAB character followed by a nonzero numeric character may be used to indicate a continuation line.

DOS/BATCH FORTRAN normally allows up to five continuation lines; however, this number may be changed to any number in the range Ø to 99 through the use of the /CO switch (see Appendix J).

1.4.4 Comment Lines

A comment may be inserted into a FORTRAN program by making the first character of the line a C. If the C character is detected in column 1 of a card or as the first character of any input record, all information on that line is disregarded. A comment line must not precede a continuation line. Continuation lines cannot be used to continue comments. All comment lines begin with the letter C.

1.5 PROGRAM UNIT STRUCTURE

A FORTRAN program unit is a FORTRAN main program or a subprogram. Except for BLOCK DATA subprograms, which are a special class (described in Section 7-6.4.8), program units are constructed of FORTRAN statements in the following order:

a. Subprogram statement, if the program unit is a subprogram.

b. IMPLICIT statement, if used.

c. Specification statements (other than IMPLICIT). Those specification statements that determine the storage required by a variable or array must precede statements that initialize the values of the variable or array. DATA statements must follow all other specification statements.

d. Arithmetic statement function definitions, if used.

e. Executable program statements. Each program unit (except BLOCK DATA subprograms) must have at least one executable statement.

f. END statement.

PART 7

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

2.1 FORTRAN CHARACTER SET

The character set from which statements can be constructed is as follows.

the letters A through Z
the digits Ø through 9

	blank	/	slash
=	equal sign	(	left parenthesis
+	plus sign	)	right parenthesis
-	minus sign	,	comma
*	asterisk	.	decimal point
$	dollar sign		

The TAB character can be used only at the beginning of a line to indicate spacing to the continuation or statement field. Other printing characters (except ↑) may appear within a FORTRAN program only as part of a comment line or a Hollerith constant (text string).

While the PDP-11 Monitor provides device independence for input to the FORTRAN Compiler, it is mandatory that FORTRAN source programs be in ASCII code. Programs intended for compilation by the DOS/BATCH FORTRAN Compiler and prepared on cards, paper tape, or magtape from other systems must conform to this requirement.

2.2 FORTRAN CONSTANTS

A constant is a self-defining value that does not change from one execution of the program to the next. In general, constants can be numeric, logical, or character string entities.

Types of constants allowed in DOS/BATCH FORTRAN include:

integer
real
double precision } - numeric constants
octal
hexadecimal
complex

logical	-	logical constants
Hollerith	-	character string constants

Storage formats for each type of constant are described in Chapter 7-12.

2.2.1 Integer Constants[1]

An integer constant is a string of numeric characters, from one to five digits in length; it is interpreted as a decimal integer. Integers are 16-bit signed quantities. A negative integer is preceded by a minus sign; a positive integer is optionally preceded by a plus sign. An integer constant must lie within the range -32768 to +32767 (which is -2^{15} to 2^{15} -1).

If the result of any operation exceeds 16 bits, the Object Time System displays an error diagnostic.

Examples of integers acceptable to DOS/BATCH FORTRAN follow.

```
3564
+14
-26357
```

Commas and decimal points are not allowed in integer constants.

2.2.2 Real Constants

A real (or floating-point) constant is a string of numeric characters including a decimal point. A real constant may contain any number of digits, but only the leftmost eight digits are significant. A negative value is preceded by a minus sign; for positive values, the plus sign is optional.

A real constant may alternately be expressed in scientific notation with a decimal exponent, represented by the letter E, and a signed integer constant. The decimal point with the real number is optional if there is an exponent. The exponent field cannot be blank, but may be zero. For positive exponents, the plus sign is optional.

A real constant has 24 bits of precision, or somewhat more than seven decimal digits. The magnitude must lie within the range Ø.14*1Ø^{38} to 1.7*1Ø^{38}. Real constants occupy two words of PDP-11 storage (see Chapter 7-12.)

[1] See Appendix J for the effect of the /ON switch on the storage allocation of integer values.

Examples of real constants follow.

```
 15.
   .579
  Ø.Ø
-1Ø.685
  5.ØE3     (i.e., 5ØØØ.)
  4.2E+3    (i.e., 42ØØ.)
  2.ØE-3    (i.e., .ØØ2)
  5EØ       (i.e., 5.Ø)
```

2.2.3 Double-Precision Constants

A double-precision constant is a string of numeric characters including a decimal point and the letter D followed by an integer exponent. The decimal point is optional. A double-precision constant may contain any number of digits, but only the leftmost 16 digits are significant. A double-precision constant is always expressed in scientific notation with a decimal exponent (represented by the letter D and a signed integer constant). The exponent field cannot be blank, but may be zero. The plus sign preceding a positive constant and/or exponent may be omitted.

A double-precision constant has 56 bits of precision, or about 16 decimal digits. The magnitude of a double-precision constant must lie between .14*1Ø$^{-38}$ and 1.7*1Ø^{38}. Double-precision constants occupy four words of PDP-11 storage. Examples of double-precision constants are shown below.

```
24.6713264837465DØ
 2.5D2                         (i.e., 25Ø. to 16 digits precision)
 3.6D-2                        (i.e., Ø36 to 16 digits precision)
 5.6DØ                         (i.e., 5.6 to 16 digits precision)
```

2.2.4 Octal Constants

The general form for an octal constant is a string of one to six octal digits (Ø to 7 inclusive) preceded by a quotation mark ("). For example:

```
"12Ø
"Ø
"177777
```

Octal constants of this form may appear anywhere an integer constant is valid. An octal constant may also be a string of one to six octal digits preceded by the letter O. Octal constants of this form are valid only in DATA, STOP and PAUSE statements.

Examples are shown below.

```
O12Ø
OØ
O177777
```

The maximum value of an octal constant is 177777.

2.2.5 Hexadecimal Constants

A hexadecimal constant is a string of one to four hexadecimal digits (Ø to 9, and A to F, inclusive) preceded by the letter Z. The hexadecimal digits follow.

Hexadecimal Digit	Decimal Equivalent
Ø	Ø
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	1Ø
B	11
C	12
D	13
E	14
F	15

A hexadecimal constant is valid only in a DATA statement. The maximum value of a hexadecimal constant is FFFF. Some examples:

```
ZA34
ZC1
ZFFFF
```

2.2.6 Complex Constants

A complex constant is written as an ordered pair of real constants separated by a comma and enclosed in parentheses. The first constant represents the real part of the complex number; the second represents the imaginary part. Each may be signed. The parentheses are part of the constant and must appear. A complex constant is represented internally by four consecutive words of PDP-11 storage. Examples follow.

```
(.7Ø712,-1.7Ø712)
(8.763E3,2.297)
```

2.2.7 Logical Constants[1]

The two logical constants .TRUE. and .FALSE. are represented internally by the integer values -1 and Ø, respectively. The delimiting periods are part of the constant and always appear. Any nonzero value is considered true when tested by a logical IF statement; -1 is equivalent to the octal number 177777. These values may be entered via DATA statements as .TRUE. and .FALSE. (See Section 7-5.7). (See Section 7-7.1Ø concerning input and output of logical values.)

Both arithmetic and logical operators may be used with logical constants, which are treated as one-word integer values.

2.2.8 Hollerith Constants

A Hollerith (or literal) constant is a string of up to 255 characters, which may be represented in one of two ways.

a. An integer constant indicating the number of characters, followed by an H, followed by a character string:

```
5HWORDS
4H1␢23
```

b. A character string enclosed in single quote characters:

```
'WORDS'
'1␢23'
```

The single quote (apostrophe) character may be included in the character string if it is immediately preceded by another single quote character. For example:

```
'DON''T'
```

Hollerith constants may be entered in DATA or input statements as a string of one or two ASCII characters per integer variable, one to four characters per real variable, and one to eight characters per complex or double-precision variable. Hollerith constants are stored in memory as byte strings, filling up to word boundaries. If a Hollerith constant has an odd number of characters, a blank is appended to the end of the constant.

[1] See Appendix J for the effect of the /ON switch on the storage allocation of logical values.

A Hollerith constant is treated as an integer quantity when it appears in an arithmetic context. For example:

```
    I = 'ABC'    +34
```

is equivalent to

```
    I = 'AB'     +34
```

The third and any subsequent characters are ignored. The integer value of 'AB' can be determined by examining the storage format, which is shown here.

15	Ø
ASCII value of B	ASCII value of A

2.2.9 Radix-5Ø Constants

Radix-5Ø is a special character data representation in which up to three characters from a limited set of characters may be packed into a single PDP-11 word. Radix-5Ø constants are allowed only in DATA statements. They are represented by a leading R followed by up to six characters and may be used to initialize only variables of type REAL.

A Radix-5Ø constant terminates with the first non-Radix-5Ø character or the sixth Radix-5Ø character, whichever comes first. Examples are shown below.

```
    R123456
    RƀƀƀƀƀA
    R1.$
```

If i, j, and k are three characters from the Radix-5Ø set, they are combined into a single word value using the formula:

$$((i * 5\text{Ø}_8 + j) * 5\text{Ø}_8 + k)$$

The Radix-5Ø characters and their code values follow.

Character	Radix-5Ø Value (Octal)
Space	Ø
A - Z	1 - 32

Character	Radix-5∅ Value (Octal)
$	33
.	34
(not used)	35
∅ - 9	36 - 47

Note that leading and embedded blanks are significant if they are within the 6-character limit, since blank is a valid Radix-5∅ character. Since the Radix-5∅ code value blank is zero, trailing blanks may be omitted without affecting the Radix-5∅ value.

2.3 FORTRAN VARIABLES

A simple FORTRAN variable is a named quantity whose value may change during execution of a program. A variable name is a string of one to six alphanumeric characters, the first of which must be alphabetic. Variable names longer than six characters are rejected by the FORTRAN Compiler, and cause an error diagnostic to be printed.

Legal Names	Illegal Names	
ALPHA	2A	(begins with a number)
MAX	MAXIMUM	(more than 6 characters)
A34		

A variable has a principal attribute type, just as constants are of one type or another. The type of variable indicates the type of value it can be assigned, its precision, and its storage requirements.

Type INTEGER, REAL, LOGICAL, DOUBLE PRECISION or COMPLEX is assigned by means of an explicit type declaration statement (see Section 7-5.2) or by an IMPLICIT statement (see Section 7-5.1). INTEGER or REAL may also be assigned by virtue of the initial letter of the variable name.

In the absence of any type declarations or IMPLICIT statements, a variable name that begins with I, J, K, L, M, or N is assumed to represent an integer variable. All other initial letters indicate a real variable.

Integer Variables	Real Variables
MAX	REAL
IDAT	DATA1
NJOB	X

2.3.1 Integer Variables

Integer variables undergo arithmetic calculations with automatic truncation of any fractional part. For example, if the current value of K is 5 and the current value of J is 9, J/K yields 1 as a result. Integer values can be converted to real values by the FLOAT function or by an arithmetic assignment statement. Integer arithmetic operations do check for overflow and print an error message if the result of an integer operation overflows 16 bits.

2.3.2 LOGICAL*1 Variables

Data items in LOGICAL*1 format are eight bits long (1 byte), and are always accessed by means of a variable name defined as BYTE or LOGICAL*1 type. The range of numbers from +127 to -128 can be represented. Any arithmetic operation is performed internally by taking two 8-bit operands, extending the sign to a full word, and performing the desired operation. No diagnostics are issued for byte overflow; the result is truncated to an 8-bit quantity.

Logical and masking operations are performed with one byte of data at a time. LOGICAL*1 (or BYTE) array elements are stored in adjacent bytes.

2.3.3 Array Variables

The extent of a variable refers to the number of values that are represented by a single variable name. A scalar variable represents a single quantity; e.g., BETA represents a single numeric value.

An array variable represents a one, two, or three dimensional array of quantities. A given variable name cannot be used as both a simple variable and an array name in the same program unit. An array element is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list can be any one, two, or three arithmetic expression(s) separated by commas.

An array can be defined with a DIMENSION or COMMON statement or as part of a type declaration statement. Array dimensioning is discussed in Section 7-5.3.

Any legal arithmetic expression can be used as a subscript of any array element once the array has been defined. Noninteger subscripts are converted to integers by truncation of the fractional part before use.

The extent of any array variable is determined by the number of dimensions it is assigned. For example, the following are legal references to array elements.

one dimensional array	(A1)
two dimensional array	REL(24,3)
three dimensional array	TR4(X*2,5,2*X)

Arrays are stored in contiguous storage locations that are addressed in ascending order with the first subscript varying most rapidly. For instance, the 2-dimensional array B(J,K) is stored in the following order:

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

See Section 7-5.3.1 for a detailed description of array storage.

References to an array variable must contain the number of subscripts with which the array was defined, the exception being those statements that can refer to an entire array given only the array name.

2.4 FORTRAN EXPRESSIONS

FORTRAN expressions can be arithmetic, relational, or logical. Relational or logical expressions have logical values that can be treated as integers. All expressions are combinations of elements and operators. The different types of expressions are discussed below.

2.4.1 Arithmetic Expressions

An arithmetic expression is a combination of arithmetic operators and elements. The arithmetic operators follow in descending order of priority.

Operator	Meaning
**	Exponentiation (A**2)
+,-	unary operators (algebraic signs: +3,-1)
*,/	multiplication (A*B), division (1/2)
+,-	addition (A+B), subtraction (D-4)

An arithmetic expression cannot contain adjacent arithmetic operators. An element preceded by a unary operator must therefore be separated from a preceding binary operator by parentheses, as in A + (-B).

In cases where two operations within an expression have the same priority (*,/ and +,-) the operations are performed from left to right. For example:

```
-A**2+B*C-1
```

is performed as

```
(-(A**2)) + (B*C) -1
```

Additional computations (such as sine, cosine, or square root) can be specified via library function references. A function reference acts as a basic element in an expression since all functions return a single value. (For example, SQRT(4.Ø) returns the value 2.Ø in an expression.)

An arithmetic expression may consist of a single element:

```
B4
22.7
NEX(3)
```

or a combination of arithmetic operators and basic elements.

```
-X+3
A/2+DAT
TAN(PI*M)
B**(-2)
```

2.4.2 Use of Parentheses

Any arithmetic expression can be enclosed in parentheses and considered as a basic element (it is evaluated before the remainder of the expression). Parentheses can be used to change the order (and the result) of evaluation. For example:

```
4+3*2-6/2 = 4+(3*2)-(6/2) = 4+6-3 = 7
(4+3)*2-6/2 = (7*2)-(6/2) = 14-3 = 11
(4+3*2-6)/2 = (4+(3*2)-6)/2 = (4+6-6)/2 = 4/2 = 2
((4+3)*2-6)/2 = ((7*2)-6)/2 = (14-6)/2 = 8/2 = 4
```

Within parentheses, the order of operator evaluation is as follows:

a. function evaluation
b. exponentiation
c. unary minus
d. multiplication, division
e. addition, subtraction

Parentheses cannot be used to imply multiplication. That is:

```
A(2)
```

is the second element of the array A, whereas

```
A*2
```

indicates twice the value of the variable A.

2.4.3 Mixed Mode Arithmetic Expressions

With few exceptions, any type of constant, variable or function can be combined with any other type in an arithmetic expression. The type of the resultant expression when any two types are combined is found in Table 7-1.

Logical, octal and Hollerith constants are treated as integer constants when combined with other elements in arithmetic expressions. The result of such a combination is then changed to the appropriate resultant mode.

Arithmetic operations can be performed between complex and other data types. Real, integer, and double-precision values (represented by V) are converted, if necessary, to the complex value (V,Ø.Ø) prior to the arithmetic operations; V is a real value in this representation.

Table 7-1

Mixed Mode Arithmetic Results

	Byte Logical*1	Integer	Logical	Real	Double Precision	Complex
Byte Logical*1	Byte	Integer	Logical	Real	Double Precision	Complex
Integer	Integer	Integer	Integer	Real	Double Precision	Complex
Logical	Logical	Integer	Logical	Real	Double Precision	Complex
Real	Real	Real	Real	Real	Double Precision	Complex
Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Complex
Complex	Complex	Complex	Complex	Complex	Complex	Complex

The general rule for mixed modes is that the result has the type of the most significant operand.

The following special rules apply for determining the type resulting from expressions of the form A**B:

1. If B is integer, the expression is of the same type as A;
2. If A and B are both Real, the expression is real;
3. If A or B is Double-precision, the expression is Double-precision;
4. No other possibilities for A and B are legal.

When a real number is converted to Double-precision format, repeating decimals are not carried through the additional precision.

The only illegal mixed mode operations are

```
B**X
I**R
I**D
C**R
C**D
X**B
X**C
```

where

B is a Byte value.

I is an Integer value.

R is a Real value.

C is a Complex value.

D is a Double-precision value.

X is a value of any data type.

2.4.4 Logical Expressions

A logical expression combines constants, logical variables, logical function references and arithmetic expressions using relational or logical operators. Logical masks can be represented by using octal constants. The logical constants .TRUE. and .FALSE. can be used in any logical expression and have the values -1 and Ø, respectively. The result of a logical expression is the logical value true or false and uses one word of storage space. (Any nonzero integer value is considered true when tested by a logical IF statement.)

A relational expression consists of two arithmetic expressions separated by a relational operator. The arithmetic expressions can be of any complexity and any combination of integer, real, or double-precision types. The relational operators follow (where A and B are arithmetic expressions).

Relational Operator	Example	Meaning
.GT.	A.GT.B	A is greater than B
.GE.	A.GE.B	A is greater than or equal to B
.LT.	A.LT.B	A is less than B
.LE.	A.LE.B	A is less than or equal to B
.EQ.	A.EQ.B	A is equal to B
.NE.	A.NE.B	A is not equal to B

Relational expressions can be written comparing values of different precision (having different data types). All lower-precision values are converted to the highest precision format. The following should be noted:

a. When combined with, or related to, a complex value, a real, integer, or double-precision value (represented by V) is converted to the complex value (V,Ø.Ø). V is a real value in this representation.

b. When a real number is converted to double-precision format, repeating decimals are not carried through the additional precision. This causes a real Ø.333..., for instance, to be less than a double-precision Ø.333... .

Only the relational operators .EQ. and .NE. can be used with complex expressions (complex quantities are equal if their corresponding parts are equal).

The value of a relational expression is always either true or false (-1 or Ø, no other value). The following are relational expressions.

```
A.GT.B
A**2+C.LT.NET*B+4
AB/C.LE.AB/D
```

Parentheses can be used in relational expressions. For example:

```
(A+B)*C.EQ.(D+1)**2
(A(1)+1).GT.(B(1))
```

A logical expression may also contain logical operators which operate on relational expressions. The logical operators are as follows.

Logical Operator	Example	Meaning
.NOT.	.NOT.A	Has the value .TRUE. only if A is .FALSE.; has the value .FALSE. if A is .TRUE.
.AND.	A.AND.B	Has the value .TRUE. only if A and B are both .TRUE.; has the value .FALSE. if either A or B is .FALSE.
.OR.	A.OR.B	Has the value .TRUE. if either A or B or both are .TRUE.; is .FALSE. only if both A and B are .FALSE. (inclusive OR).

A logical expression may consist of basic logical elements or a combination of logical elements. For example, the following are all logical expressions (relational expressions are a subtype of logical expressions).

```
.TRUE.
X.GE.3.14159
TV(1).AND.INDEX
B(M).OR.(K.EQ.LIMIT)
```

No two relational or logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators. Any logical expression can be preceded by the unary operator .NOT. as follows.

```
.NOT.T
.NOT.(X+7.GT.Y+Z)
B(M).AND..NOT.TV.OR.R    which is the same as   (B(M).AND.(.NOT.TV)) .OR.R
```

However, where .NOT. appears next to an arithmetic operator, the arithmetic expression must appear in parentheses. For example:

```
.NOT.(+1.Ø).LE.DA(2)
```

as opposed to the following form, which is incorrect:

```
.NOT.+1.Ø.LE.DA(2)
```

Logical and relational operators have priorities in the following order when the precedence is not established by parentheses.

```
.GT.   .GE.   .LT.   .LE.   .EQ.   .NE.   .NOT.   .AND.   .OR.
```

Relational operators are evaluated in an expression from left to right subject to the priority rules above. In a logical expression, all arithmetic evaluation is performed prior to any logical evaluation. Thus, the logical expression

```
.NOT.Z**2+Y*MA .GT. K-2 .OR. PA .AND. X .EQ. Y
```

is interpreted as

```
(.NOT.(((Z**2) + (Y*MA)).GT.(K-2))).OR.(PA.AND.(X.EQ.Y))
```

Parentheses may be used within logical expressions to make an expression more readable or to override established priorities. For example, the following two expressions are evaluated differently.

```
A.AND.(B.OR.C)
(A.AND.B).OR.C
```

2.5 OPERATOR SUMMARY

Operators in each type are shown in order of descending priority.

Type		Operator	Operates Upon
arithmetic	**	exponentiation	numeric constants variables and expressions
	+,-	unary plus, minus	
	*,/	multiplication, division	
	+,-	addition, subtraction	
relational	.GT.	greater than	logical variables, logical constants and arithmetic expressions (all relational operators have equal priority)
	.GE.	greater than or equal to	
	.LT.	less than	
	.LE.	less than or equal to	
	.EQ.	equal to	
	.NE.	not equal to	
logical	.NOT.	.NOT.A is true if and only if A is false; it is false if A is true.	logical variables and logical constants
	.AND.	A.AND.B is true if and only if A and B are both true; it is false if either A or B is false.	
	.OR.	A.OR.B is true if either A or B or both are true; it is false if both A and B are false.	

PART 7

CHAPTER 3

ASSIGNMENT STATEMENTS

There are three types of assignment statements.

1. Arithmetic assignment statements
2. Logical assignment statements, and
3. ASSIGN statements.

A value can be assigned to a variable at any point in a source program. During program execution, the most recent assignment determines the value of that variable in subsequent statements.

3.1 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement provides the means by which the results of computations can be stored.

The format of the arithmetic assignment statement is

```
     variable = expression
```

where variable is any legal simple or subscripted variable name, expression is any legal arithmetic expression, and = is the replacement operator. The equal sign does not indicate equality as in an algebraic statement, but rather a replacement. The value of the expression is assigned to the variable name, regardless of any previous value of the variable. For example,

```
     I = I+1
```

takes the current value if I; adds 1 to it, and uses the new value as the current value of I for any subsequent operations.

The expression to the right of the equal sign is evaluated and converted when necessary to conform to the type of the variable to the left of the equal sign. That is, if a real expression is assigned to an integer variable, the value of the expression is converted (truncated) to an integer before assignment. For example:

```
     AND = Y*(X**2+Z)
     I = I*N
```

```
X(J) = (J)-B(J)
J = A**2+C
```

The expression to be assigned must be capable of yielding a value that conforms to the type attribute of the variable that is to receive that value. Type conversions are performed in accordance with the rules stated for mixed mode expressions in Section 7-2.4.3 and Tables 7-1 and 7-2.

Table 7-2

Conversions Rules for Assignment Statements

	Expression Type				
Variable Type	REAL	INTEGER LOGICAL, Literal or OCTAL Constant	COMPLEX	Double PRECISION	Byte LOGICAL*1
REAL	Store X in V	Make X REAL store in V	Store REAL part of X in V imaginary part lost	Round to REAL and store in V, low-order portion lost	Sign-extend byte to INTEGER convert INTEGER to REAL and store in V
INTEGER LOGICAL	Truncate X to INTEGER and store in V	Store X in V	Truncate REAL portion of X to integer in V, imaginary part = Ø.Ø	Truncate X to INTEGER and store in V	Store X in low-order portion of V with sign bit extended through high-order portion of V.
COMPLEX	Store X in REAL portion of V, imaginary portion = Ø.Ø	Make X REAL and store in REAL portion of V, imaginary portion = Ø.Ø	Store X in V	Round X to REAL, store in REAL part of V, imaginary part = Ø.Ø	Sign-extend byte to INTEGER, convert INTEGER to COMPLEX (imaginary part = Ø.Ø) and store in V.
DOUBLE PRECISION	Make X DOUBLE precision with low-order portion = Ø store in V	Make X REAL and store in high-order portion of V, low-order portion = Ø	Set V to REAL portion of X with low-order portion of V=Ø, imaginary part lost	Store X in V	Sign-extend byte to INTEGER, convert INTEGER to DOUBLE PRECISION, store in V.
BYTE LOGICAL*1	Convert REAL portion to INTEGER store low-order 8 bits of INTEGER in byte	Store low-order 8 bits in V	Convert REAL portion to INTEGER store low-order 8 bits of INTEGER in byte	Convert DOUBLE PRECISION portion to INTEGER, store low-order 8 bits of INTEGER in byte	Store X in V

where: X indicates the expression that is evaluated on the right of the equal sign.
V indicates the variable name to the left of the equal sign, as in the statement: X=X

3.2 LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is of the form

```
     variable = expression
```

where variable is a predefined logical variable name or a logical array element name, expression is a logical expression, and = is the replacement operator. As in the arithmetic assignment statement, the equal sign indicates that the value of the expression (true or false) is given to the variable specified. For example:

```
     P = .TRUE.
     VAL = X.GT.4.OR.B.EQ.X
     A = R.LE.X**2
```

In the case where mixed mode arithmetic expressions are being logically compared, Table 7-3 shows the mode in which the comparison is effected. For example,

```
     A = X**2 .GE.  I
```

causes X**2 to be evaluated, the value of the integer variable I converted to real format, and the comparison made.

3.3 ASSIGN STATEMENT

The ASSIGN statement is of the form

```
     ASSIGN n TO ivar
```

where n is a statement number and ivar is an integer variable. The ASSIGN statement permits a subsequently-executed assigned GOTO statement (see Section 4.1.3) to transfer control to the statement identified by the variable ivar. The ASSIGN statement and the associated GOTO statement(s) must appear in the same FORTRAN program unit. The statement number assigned must be that of an executable statement (assignment, I/O, or control statement).

An integer variable used in an ASSIGN statement must be redefined in an arithmetic assignment statement before it can be used in any context other than the GOTO statement (and after which it can no longer be used in the GOTO statement). For example,

```
     ASSIGN 10 TO COUNT
```

associates the variable name COUNT with statement number 10, after which the statement

```
      COUNT=COUNT+1
```

is considered invalid by the FORTRAN system. The above statement can be performed, however, if preceded by the statement.

```
      COUNT=2Ø
```

which assigns the value of 20 to the variable COUNT. But the variable COUNT can no longer be used to refer to statement number 10. Incorrect usage of a variable defined in an ASSIGN statement causes a warning diagnostic to be printed by the Compiler.

PART 7

CHAPTER 4

CONTROL STATEMENTS

4.1 GOTO STATEMENTS

GOTO statements allow program execution to jump from one section of code to another, either directly or based on the value of a variable.

There are three types of GOTO statements.

a. Unconditional GOTO statements
b. Computed GOTO statements
c. Assigned GOTO statements

4.1.1 Unconditional GOTO Statement

The unconditional GOTO statement transfers control directly to the specified statement number. The form of this statement is

```
      GOTO n
```

where n is the statement number of an executable statement. For example:

```
      GOTO 1ØØ
```

When the above statement is encountered, program execution continues at the statement identified by statement number 1ØØ.

An unconditional GOTO statement may appear anywhere in the executable part of a source program, except as the terminal statement of a DO Loop. The GOTO statement and the statement to which it transfers control must both appear in the same FORTRAN program unit.

4.1.2 Computed GOTO Statement

The computed GOTO statement allows the user to transfer control to one of several statement numbers, depending upon the value of a variable. The form of the statement ~~is~~ as follows:

```
GOTO(n1,n2,n3,...,nk),ivar
```

where n_i are statement numbers and ivar is an integer variable whose value is between +1 and k. The comma between the statement number list and ivar is optional. Control is automatically transferred to the statement whose number is n_{ivar} in the list.

```
GOTO (75,15Ø,1ØØ,85,275)L
```

If L is equal to 1 when the statement is executed, control transfers to statement number 75 in the program; if L equals 5, control transfers to statement number 275. If L has a value less than 1 or greater than 5 at the time this statement is executed, control passes to the next statement in the program. There is no restriction on other uses of the integer variable L within the program.

A computed GOTO statement may appear anywhere in the executable part of a source program except as the terminal statement of a DO loop. The statement numbers in parentheses must identify executable statements that exist in the same program unit as the GOTO statement.

4.1.3 Assigned GOTO Statement

The assigned GOTO statement allows the user to transfer control based on an integer variable name.

Assigned GOTO statements may take one of two forms.

```
GOTO K
```

or

```
GOTO K, (n1 ,n2 ,n3 ,...,nk)
```

where K is an integer variable name and n_i are statement numbers. Control is transferred to the statement whose number is currently associated with the variable K via an ASSIGN statement.

As described in Section 7-3.3, an ASSIGN statement defines an integer variable as a statement number. Thus, when the statement

```
ASSIGN 1Ø TO LOOP
```

is executed, the programmer can transfer control to statement 10 by inserting the statement:

```
        GOTO LOOP
```

or, alternatively:

```
        GOTO LOOP,(1Ø,2Ø,1ØØ)
```

which transfers control to whichever statement number is currently associated with LOOP. A statement number must have been assigned to LOOP prior to execution of the GOTO statement; if the optional statement list is included, the number assigned to LOOP must be one of the listed numbers. Otherwise, control passes to the next sequential statement.

The GOTO statement and the executable statement to which it transfers control must both appear in the same FORTRAN program unit.

4.2 IF STATEMENTS

IF statements allow program execution to jump from one section of code to another or conditionally execute a statement based on the value of an expression. There are two types of IF statements.

a. Arithmetic IF statements
b. Logical IF statements

Care should be taken in mixed mode comparisons or comparisons in which one or more values have undergone a type conversion. Mixed mode values to be compared are set to the more significant of the two types being compared. When this is done, repeating binary fractions do not repeat to the full precision of the new mode. For example, this may cause a real Ø.1 (1.ØE-1) to be unequal to a double-precision Ø.1 (1.ØD-1).

4.2.1 Arithmetic IF Statement

An arithmetic IF statement is of the form

IF (expression) n_1 ,n_2 ,n_3

where expression is an arithmetic expression in parentheses and n_1 ,n_2 ,n_3 represent the statement numbers to which control is transferred if the expression is less than, equal to, or greater than zero, respectively. All three statement numbers must be present (although two may refer to the same statement). For example:

```
ALPHA = -7
.
.
.
IF (ALPHA) 1Ø, 2Ø, 1Ø
```

Control transfers to statement number 1Ø, since the expression is less than zero. It would also transfer to statement 1Ø if the expression were greater than zero. Control would transfer to statement number 2Ø only if the expression were exactly equal to zero.

Expressions having complex or logical results cannot be used in arithmetic IF statements.

4.2.2 Logical IF Statement

A logical IF statement is of the form

```
IF (expression) statement
```

where expression is a logical expression in parentheses and statement is an executable FORTRAN statement other than another IF statement or a DO statement. If the expression has the value .TRUE. the contained statement is executed; otherwise, the next sequential statement is executed.

Examples of logical IF statements are shown below.

```
IF (T.OR.S) X=Y+1
IF (Z.GT.X(K)) CALL SWITCH(S,Y)
IF (M.EQ.INDEX) GOTO 15
```

4.3 DO STATEMENT

The DO statement simplifies the coding of repetitive procedures. The DO statement is of the form

DO n i = m_1 ,m_2 ,m_3

where n is a statement number, i is a unsubscripted integer variable, and m_1 ,m_2 ,m_3 are positive integer constants or unsubscripted integer variables. If m_3 is not specified, it is understood to be 1. A zero or negative m_3 value is not permitted.

Statements following the DO statement, up to and including statement n, are executed repeatedly for values of i starting with m_1 and incremented by m_3 until i is greater than m_2.

The statements that are executed as a result of a DO statement are called the range of the loop. The variable i is called the index variable. The terms m_1 ,m_2 ,m_3 are the initial, terminal, and increment parameters of the index, respectively. The initial parameter value must be less than or equal to the terminal parameter value.

When the DO statement is executed, its range is first executed with the index equal to the initial parameter's value. After each execution of the range, the increment value is added to the value of the index and the result compared with the terminal parameter value. If the index value is not greater than the terminal parameter value, and integer overflow has not occurred, the range is executed again with the new index value. The range of a DO loop is always executed at least once.

For example:

```
DO 20 I = 5,1ØØ,2            (final iteration has I = 99)
DO 1ØØ M = 1Ø,1ØØ,2          (final iteration has M = 1ØØ)
```

After the last execution of the range, control passes to the statement immediately following the loop. This is tne normal exit from a DO loop. Exit can also be accomplished by the execution of a control statement within the range. (Control cannot be transferred into the range of a DO loop from outside that loop except in the case of an extended range as described below.)

The values of the terminal and increment parameters, and of the index variable, of the DO loop cannot be altered within the range of the DO statement. The index variable is, however, available for use as an ordinary variable within the range of a DO loop, with a GOTO or IF, the index variable retains its current value and is available for use as a variable. The value of an index variable is not defined on normal edit from the DO loop.

The terminal statement of a DO loop cannot be a GOTO, DO, RETURN, or arithmetic IF statement. A logical IF statement is allowed as the last statement of the range, provided it does not contain any of the statements mentioned above. For example:

```
        DO 5 K = 1,4
  5     IF (X(K).GT.Y(L))  Y(K)=X(K)
  6     ...
```

In this case, the loop ends when control passes to statement number 6. Statement 5 is executed four time whether or not the Y(K)=X(K) statement is executed. Note that if statement 5 were

```
  5     IF (X(K).GT.Y(L)) GOTO 1Ø
```

improper execution may result.

The range of a DO statement may include other DO statements, provided the range of each DO statement is contained (nested) entirely within the range of the next outermost DO statement. That is, the range of any two DO statements must intersect completely or not at all. More than one DO loop within a nest of DO loop can end on the same terminal statement.

The range of a DO loop need not be merely a section of straight line code following the DO statement. A control statement that causes execution of instructions elsewhere in the program is permissible as long as control returns to the range of the originating loop. The statements executed between the pair of control statements (leaving and returning to the DO loop) are called the extended range of the loop.

DO loops can be nested up to 10 levels deep. In calculating this depth, an implied DO in an I/O statement (see Section 7-9.1.3) counts as one level, whose range is a single statement; and n implied DO loops within one I/O statement count as n levels whose ranges are all within the single statement. For example,

```
  5Ø WRITE (5,25Ø) (BUF,F(I),I=1,5),(DIFF(I),I=1,3)
```

contains two complete implied loops resulting in one level of nesting at a time. If the above statement were contained in an unnested DO loop, the maximum nesting level generated by line 50 would be two.

NOTE

> A common typing mistake is to replace a comma in a DO statement with a period. For example:
>
> ```
>       DO 5 I = 1.5
> ```
>
> This statement is interpreted by the FORTRAN Compiler as an arithmetic assignment statement:
>
> ```
>          DO5I = 1.5
> ```

4.4 CONTINUE STATEMENT

The CONTINUE statement is of the following form.

```
      CONTINUE
```

No processing is done when this statement is executed. It is used primarily as a target line number for control transfers, particularly as the terminal statement of a DO loop. For example:

```
      DO 7 K = START,END
      .
      .
      .
      IF (X(K)) 22,13,7
      .
      .
      .
  7   CONTINUE
```

A positive value of X(K) begins another execution of the range or causes a normal exit from the loop.

In this example, the CONTINUE statement provides a target address for the IF statement and ends the range of the DO loop.

The CONTINUE statement can be inserted anywhere in the executable portion of a program with no effect.

4.5 PAUSE STATEMENT

The PAUSE statement takes one of the following forms.

```
      PAUSE
```

or

```
      PAUSE n
```

where n is an octal constant from one to six digits long (up to 177777) optionally preceded by the letter O. The PAUSE statement temporarily interrupts program execution.

4.6 STOP STATEMENT

The STOP statement is of the form

```
STOP
```

or

```
STOP n
```

where n is an octal constant from one to six digits long (optionally preceded by the letter O). The STOP statement prints the constant (zeros, if no constant is specified), terminates execution of the current program, and returns control to the Monitor.

4.7 END STATEMENT

The END statement is of the following form.

```
END
```

This statement must be the last statement in every FORTRAN program unit. Program compilation is terminated when the END statement is encountered in a main program unit.

An END statement forces a call to the EXIT subroutine in a main program. A function or subroutine subprogram forces a return to the calling program if a RETURN statement is not present.

If a compiler input file is exhausted before an END statement is encountered, a diagnostic is printed and an END statement forced.

PART 7

CHAPTER 5

SPECIFICATION STATEMENTS

Specification statements are nonexecutable statements generally placed at the beginning of a program to inform the FORTRAN Compiler about storage allocation, variable types, subprograms, or data.

Specification statements must be placed in the program prior to the first usage of any variable they describe. Some specification statements determine storage allocation and must appear prior to any executable statements. Those statements that must precede all executable statements are called declaratives and include:

```
IMPLICIT
EXTERNAL
type declaration
COMMON
EQUIVALENCE
DIMENSION
DATA
```

The IMPLICIT declaration, if present, should occur first as its action may affect any other declaratives. DATA statements, if used must appear last, since other declaratives may have an effect on the data elements they define.

5.1 IMPLICIT STATEMENT

The IMPLICIT statement is of the form

IMPLICIT type $a_1,a_2,(b_1-b_2),a_3,(b_3-b_4),\ldots$

where type indicates the variable type being defined (see Section 7-5.2 for a complete list). The elements a_i represent single alphabetic characters separated by commas. The elements (b_n-b_m) represent a range of characters denoted by the first and last characters of the range and separated by a minus sign. A range of characters is always enclosed in parentheses. For example:

```
IMPLICIT INTEGER (A-D), (X-Z)
```

This statement instructs the FORTRAN Compiler that any variable name in the program beginning with the letters A, B, C, D, X, Y, or Z is an integer variable.

The effect of the IMPLICIT statement is to extend the basic rule that all names beginning with I, J, K, L, M or N indicate integer variables; all others are real variables. Any variable name that does not appear explicitly in a type declaration statement and whose first character is one of those listed in an IMPLICIT statement is classified according to the type appearing in that IMPLICIT statement.

The initial state of the FORTRAN Compiler is as though the statements

```
        IMPLICIT REAL(A-H),(O-Z)
        IMPLICIT INTEGER(I-N)
```

appeared at the beginning of each program. This state is in effect unless a type declaration or IMPLICIT statement changes the interpretation of any variable name. Definition of a variable name in a type declaration statement overrides any type specification determined by an IMPLICIT statement.

The IMPLICIT statement, if present in a subprogram, must precede all other statements (except the SUBROUTINE and FUNCTION statement). The IMPLICIT statement becomes effective when it is encountered in the program. Previous specifications of variables having implicitly defined values will cause a diagnostic. For example,

```
        COMMON Y1

        IMPLICIT INTEGER (X-Z)
        .
        .
        .
```

results in a W (warning) diagnostic.

Most of the FORTRAN Library function names (FLOAT, IFIX, etc.) follow the default type rules. That is, FLOAT produces a real result, IFIX produces an integer result, and so on. The use of an IMPLICIT statement may change the default type, causing undesirable results. For example, in the program,

```
        IMPLICIT INTEGER (A-Z)
        REAL X
        I=1
        X=FLOAT(I)
        END
```

the function FLOAT is treated as though it produced an integer result leading to incorrect values being generated. To have the FLOAT function return a real value in this context, an explicit type declaration should be included in the program before or after the IMPLICIT statement, as follows.

```
        REAL FLOAT
```

Those library functions whose names do not follow the default type rules (those that return complex or double-precision values) are recognized by the compiler under any conditions. The data type of these names is not affected by an IMPLICIT statement but may be changed by an explicit type declaration. For example:

```
        INTEGER CABS
```

This statement could cause incorrect operations at execution time if the intrinsic function CABS is called.

5.2 TYPE DECLARATION STATEMENT

A type declaration statement provides the compiler with a description of the storage format of the variables defined, and is of the form

type $\text{var}_1,\text{var}_2,\ldots$

where type indicates the name of the variable type being defined. Acceptable types:

INTEGER	(1- or 2-word integer numbers)
INTEGER*2	(same as INTEGER)
REAL	(2-word real numbers)
REAL*4	(same as REAL)
DOUBLE PRECISION	(4-word real numbers)
REAL*8	(same as DOUBLE PRECISION)
COMPLEX	(4-word values, each having a 2-word real part and a 2-word imaginary part)
LOGICAL	(true = -1, false = Ø)
BYTE	(8-bit format)
LOGICAL*1	(same as BYTE)

Allocation of two bytes for 1-word integers is effected only by the /ON switch. INTEGER*2 causes four bytes per element when the /ON switch is not set (see Section 7-12.1).

Allocation of two bytes for LOGICAL variables is effected only by the /ON switch. LOGICAL allocates four bytes per element when the /ON switch is not set (see Section 7-1Ø.2).

The terms var_1,var_2,... indicate the variable name(s) that are to be allowed storage space in the data type format specified.

Type declarations must precede all executable statements. For example:

```
COMPLEX RET,X,TST
BYTE IAMB,LEN,R(2Ø),S(2Ø)
```

In the first statement, three variable names are specified as complex variables. In the second statement, two variable names are specified as 8-bit byte variables, and the arrays R and S are dimensioned as having byte-sized elements each.

Type declarations can be used to define dimension specifications for arrays. However, an array defined in a type declaration cannot be redimensioned by any other specification statement, and vice versa. Adjustable arrays in subprograms can also be defined in type declarations.

If a variable name is defined in a type declaration, the type so specified overrides any type specification determined by the initial letters of the variable. The priority for defining a variable type is, therefore, as follows:

a. type declarations.
b. IMPLICIT variable type.
c. real or integer, depending on the initial letter (where I through N indicates an integer).

5.3 DIMENSION STATEMENT

The DIMENSION statement is of the form

DIMENSION S_1,S_2,S_3,...

where S is an array specification of the form

$var(n_1)$
$var(n_1,n_2)$

or

$var(n_1,n_2,n_3)$

where var is the array variable name, and n_1, n_2, and n_3 are unsigned integer constants (or variables, optionally, in the case of adjustable array dimensions in subprograms) indicating the maximum size of the array in each of the three possible dimensions.

An array can have one, two, or three dimensions. For example:

```
      DIMENSION A(2Ø), B(1Ø,1Ø), C(5,5,5)
```

Where more than one array is defined in a single DIMENSION statement, the array specifications are separated by commas.

DIMENSION statements must precede all executable statements.

The appearance of the DIMENSION statement causes the FORTRAN Compiler to allocate space for the appropriate number of variables. Array definition can also be performed by a COMMON statement or any type declaration statement. For example:

```
      COMMON X(1Ø,4),Y,Z
      INTEGER A(7,32)
```

Once defined, an array cannot be redimensioned by any other specification statement.

Unless an array is specified in a type declaration statement, the type of the array is determined by the first letter of the array variable name.

An error message is always generated at run-time if an array subscript is less than one. Optionally, using the /CK switch option (see Appendix J), the system will check to see that array subscripts do not exceed the maximum size specified.

A subprogram can use adjustable arrays, the size of which may depend upon the parameters of the subprogram call (see Sections 7-6.4.1 and 7-6.4.2).

5.3.1 Array Storage

Arrays of one, two, or three dimensions are each stored as a linear sequence of values in memory. Arrays are always stored in memory column by column. Therefore, the n^{th} element of a 1-dimensional array is A(n).

In order to find the linear element of a 2-dimensional array A(I,J) where a given element is A(i,j), the formula

```
      n = i + I*(j-1)
```

is evaluated. A(i,j) is stored at the n^{th} element.

In the 3-dimensional array (A(I,J,K) where a given element is A(i,j,k), an individual element is stored at the n^{th} element, where n is

```
      n = i + I*(j-1) + I*J (k-1) .
```

It is sometimes desirable to equivalence a 2- or 3-dimensional array to a 1-dimensional array. The above formula is useful for determining the algorithms for manipulating the arrays after such a change. However, for any given array, elements of that array must be referenced using the number of subscripts specified in the defining statement (although in some contexts an entire array can be referenced by indicating the array name only). See Figure 7-2.

5.4 COMMON STATEMENT

The COMMON statement causes the specified variables to be stored in an area of memory available to other programs and subprograms. Common block areas can be of any size and any number of them can exist within the space available on the system. One unnamed common block (known as blank common) is allowed. The variables are placed in the individual common blocks in the order in which they appear in the block description.

The COMMON statement takes the form

```
      COMMON /block1/a,b,c/block2/d,e,f
```

where blockn is a 1- to 6-character alphanumeric name (the first character of which must be alphabetic), delimited by slashes, and considered to be the block name. The sequences a,b,c and d,e,f represent the names of the variables assigned to the common block(s). The form

```
     COMMON a,b,c,...,n
```

is used to indicate the blank common block.

One or more common blocks can be defined in the same COMMON statement.

For example:

```
      COMMON /A/X,Y,Z/B/L,M,N
```

1-Dimensional Array A(1Ø)

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(1Ø)

2-Dimensional Array B(5,5)

1	B(1,1)	6	B(1,2)	11	B(1,3)	16	B(1,4)	21	B(1,5)
2	B(2,1)	7	B(2,2)	12	B(2,3)	17	B(2,4)	22	B(2,5)
3	B(3,1)	8	B(3,2)	13	B(3,3)	18	B(3,4)	23	B(3,5)
4	B(4,1)	9	B(4,2)	14	B(4,3)	19	B(4,4)	24	B(4,5)
5	B(5,1)	1Ø	B(5,2)	15	B(5,3)	2Ø	B(5,4)	21	B(5,5)

B(3,1) is the third storage element in sequence
B(3,4) is the eighteenth storage element in sequence

3-Dimensional Array C(5,5,5)

										1Ø1	C(1,1,5)	1Ø6	C(1,2,5)	111	C(1,3,5)	116	C(1,4,5)	121	C(1,5,5)
										1Ø2	C(2,1,5)	1Ø7	C(2,2,5)	112	C(2,3,5)	117	C(2,4,5)	122	C(2,5,5)
						76	C(1,1,4)	81	C(1,2,4)	86	C(1,3,4)	91	C(1,4,4)	96	C(1,5,4)	118	C(3,4,5)	123	C(3,5,5)
						77	C(2,1,4)	82	C(2,2,4)	87	C(2,3,4)	92	C(2,4,4)	97	C(2,5,4)	119	C(4,4,5)	124	C(4,5,5)
				51	C(1,1,3)	56	C(1,2,3)	61	C(1,3,3)	66	C(1,4,3)	71	C(1,5,3)	98	C(3,5,4)	12Ø	C(5,4,5)	125	C(5,5,5)
				52	C(2,1,3)	57	C(2,2,3)	62	C(2,3,3)	67	C(2,4,3)	72	C(2,5,3)	99	C(4,5,4)				
		26	C(1,1,2)	31	C(1,2,2)	36	C(1,3,2)	41	C(1,4,2)	46	C(1,5,2)	73	C(3,5,3)	1ØØ	C(5,5,4)				
		27	C(2,1,2)	32	C(2,2,2)	37	C(2,3,2)	42	C(2,4,2)	47	C(2,5,2)	74	C(4,5,3)						
1	C(1,1,1)	6	C(1,2,1)	11	C(1,3,1)	16	C(1,4,1)	21	C(1,5,1)	48	C(3,5,2)	75	C(5,5,3)						
2	C(2,1,1)	7	C(2,2,1)	12	C(2,3,1)	17	C(2,4,1)	22	C(2,5,1)	49	C(4,5,2)								
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)	5Ø	C(5,5,2)								
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)										
5	C(5,1,1)	1Ø	C(5,2,1)	15	C(5,3,1)	2Ø	C(5,4,1)	25	C(5,5,1)										

C(1,3,2) is the 36th storage element in sequence.
C(1,1,5) is the 1Ø1st storage element in sequence.

Figure 7-2
Array Storage

This statement defines two common blocks, each containing three variables.

Where blank common is the first common block being described in the COMMON statement, no block name or specification is necessary. Subsequent block names in the same COMMON statement are delimited by slashes. However, when elements of blank common are indicated other than in the first position in a COMMON statement, two consecutive slashes (any separating spaces are ignored) are used. For example:

```
COMMON /R/X,Y//B,C,D
COMMON B,C,D/R/X,Y
```

The two COMMON statements above are identical in effect.

All COMMON statements within a program unit must precede all executable statements.

A common variable cannot appear in more than one common block in a single program unit. However, since labeled common block names are used only by the compiler and are not present in the functional program at run time, such block names can be used as variable or subroutine names within the program.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements

```
COMMON/A/ALPHA/B/BET,GAM/C/ET
COMMON/B/DEL,EP,ZE/C/THE,IO/D/KA,KA
```

have the same effect as the statement

```
COMMON/A/ALPHA/B/BET,GAM,DEL,EP,ZE/C/ET,THE,IO/D/KA,LA
```

Storage allocation for blocks of the same name begins at the same location for all program units linked together in a single load module. For example, if a program contains the statement

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram contains the statement

```
COMMON /R/U,V,W//D,E
```

as its first COMMON statement, the quantities represented by X and U, by Y and V, and by Z and W (provided they are of corresponding word lengths) are stored at the same locations in common block R; a similar correspondence holds for the quantities A and D, and B and E, in the blank common block. Responsibility for the values of the variables used and the correspondence of word lengths of the variables rests with the programmer.

The size of a common block is the sum of the storage required for all elements introduced through COMMON and EQUIVALENCE statements. Where the size of the various common blocks referenced by several program units in a single load module differs, the length of each common block is resolved at linkage time to be the length of the longest block of the given name declared by any program unit.

The variable names that follow the block name indicate scalar or array variables assigned to that block. Where an unsubscripted array name appears in a COMMON statement, the array must have been previously defined in a DIMENSION or type declaration statement. Where a subscripted array name appears in a COMMON statement and the array has not been defined, the array is defined as being the indicated size. Where the array has been dimensioned previously, an error diagnostic is printed. For example:

```
DIMENSION T(10)
COMMON ABS(5,10,5),T
```

The COMMON statement defines the array ABS and allocates storage space for that array in blank common. All elements of the array T are also stored in blank common. (Arrays in common storage are stored according to the rules specified in Section 7-5.3.1.)

When both BYTE (LOGICAL*1) and other types of variables are included in the same COMMON block, care must be taken to assure that non-BYTE elements that follow BYTE elements are allocated on a word boundary. Non-BYTE elements may only follow an even number of BYTE elements. (Failure to assure this may result in a "word reference to odd address" hardware error when the program is executed.)

5.5 EQUIVALENCE STATEMENT

The EQUIVALENCE statement is of the form

EQUIVALENCE $(var_1,var_2,\ldots),(var_i,var_j,\ldots),\ldots$

where var indicates a variable name used in the current program unit. Each of the variables within a set of parentheses is assigned to the same storage location. For example,

```
      EQUIVALENCE (LAN,MAR)
```

specifies that the variables LAN and MAR are stored in the same location, and have the same value. Where variables being equivalenced are not of the same type, the equivalencing operation assumes that each variable begins in the same word (see the description of storage formats in Chapter 12).

EQUIVALENCE statements must precede all executable statements.

The equivalence relationship is transitive; that is, the two statements below have the same effect.

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

The EQUIVALENCE statement is used to conserve core storage during execution or to obtain a portion of a numeric value in a different format. Rather than reserving space for a variable that is only used for a short time in the program, a later variable can be equivalenced to use this space when the earlier value is no longer of interest. A statement such as the following

```
      EQUIVALENCE (TRI,ISO)
```

allows ISO to represent the storage location formerly occupied by the real variable TRI.

To make full use of the space required by a real variable, that real variable can be equivalenced to two integer variables (assuming use of the /ON switch[1] which causes 1-word integers to be used throughout the program). Similar techniques can be used to equivalence two real, or two to four integer, variables to complex or double-precision variables. For example:

```
      DIMENSION N(2)
      EQUIVALENCE (TRI,N)
```

[1] See Appendix J.

These two statements equivalence the first word of TRI to N(1) and the second word of TRI to N(2), assuming use of the /ON switch.

5.5.1 Equivalencing Array Variables

The subscripts of any array variables appearing in an EQUIVALENCE statement must be integer constants. For example:

```
EQUIVALENCE (X,A(5)),(BET(2,2),Y(2,4,1),NET)
```

A previously-dimensioned array name used in an EQUIVALENCE statement without subscripts implies reference to the first element of the array. For example,

```
DIMENSION Y(2Ø)
EQUIVALENCE (Y,X)
```

has the same effect as

```
DIMENSION Y(2Ø)
EQUIVALENCE (Y(1),X) .
```

Where a subscript is present in an EQUIVALENCE statement, the array element specification must contain an integer constant for each of the array dimensions as originally defined. No default value is supplied for missing dimension elements, and a diagnostic is generated. For example,

```
DIMENSION REAL(2,1Ø,1Ø)
EQUIVALENCE (REAL(1,1),TRAN)
```

is illegal and would result in an error message. (See also the description of array storage in Section 7-5.3.1).

5.5.2 EQUIVALENCE and COMMON Interaction

The same variable name may appear in both COMMON and EQUIVALENCE statements. However, two quantities in common cannot be made equivalent to each other. Quantities placed in a common block by an EQUIVALENCE statement may cause the end of the common block to be extended. For example,

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

causes the common block R to extend from X to A(4) as follows.

```
X
Y=A(1)
Z=A(2)
  A(3)
  A(4)
```

The end of the common block has been extended by two elements. No EQUIVALENCE statements are allowed to extend the beginning of a common block, however. For example, the sequence

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

is not permitted, since it would require A(1), A(2) and A(3) to extend beyond the starting location of common block R.

An EQUIVALENCE statement is not permitted to violate any previous EQUIVALENCE storage assignments or extend any array beyond its maximum predefined dimensions. For example:

```
DIMENSION A(1Ø),B(1Ø)
EQUIVALENCE (A(1Ø),B(1)),(A(5),B(3))
EQUIVALENCE (A(5,1Ø),B(5))
```

Both of the preceding EQUIVALENCE statements are unacceptable.

5.5.3 EQUIVALENCE and BYTE Arrays

In using EQUIVALENCE with items in a BYTE (LOGICAL*1) array, only those items aligned on a word boundary can be specified in an EQUIVALENCE statement. Individual BYTE variables are always aligned on word boundaries except within COMMON blocks (see Section 7-5.4). In a BYTE array, only the elements having odd subscripts are aligned on word boundaries. For example:

```
                        BYTE A,B(9)
                        REAL X

correct:                EQUIVALENCE(A,X,B(3))

incorrect:              EQUIVALENCE(A,X,B(4))
```

5.6 EXTERNAL STATEMENT

The EXTERNAL statement is of the form

```
      EXTERNAL a,b,c,...
```

where a,b,c,... represent the names of FUNCTION or SUBROUTINE subprograms to be used as parameters to a subprogram call.

FUNCTION and SUBROUTINE subprogram names can be used as arguments of other subprograms. When used in this manner, subprogram names are distinguished from ordinary variables by their appearance in an EXTERNAL statement. For example:

```
      EXTERNAL SIN,COS
        .
        .
        .
      CALL TRIGF(SIN,1,5,ANSWER)
        .
        .
        .
      CALL TRIGF(COS,18.7,ANSWER)
        .
        .
        .
      END

      SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
        .
        .
        .
      ANSWER = FUNC(ARG)
      RETURN
      END
```

The EXTERNAL statement must appear before any executable statements in any program or subprogram. This statement is used in programs or subprograms using the name of a function or subroutine as a parameter in a subprogram call. This statement does not take a line number and should not contain any name previously defined.

5.7 DATA STATEMENT

The DATA statement is of the form

```
      DATA var/value/,var/value/,...
```

where var is a list of variable names, and value is an ordered list of numeric or string values that are to be assigned to the respective elements of the variable list.

The DATA statement is used to supply initial or constant values for variables. DATA statements must precede all executable statements and must follow all other declaratives. Elements of the variable and value lists are separated by commas. More than one set of variable and value lists can be contained in a single DATA statement. For example:

```
DATA A,B,C/3.4,5.9,7./

DATA I,J/7,3/X,Y,Z/43.5,2.Ø1,8.4/
```

Dummy arguments must not appear in the variable list. Whenever several variables are to be assigned the same value, the item in the value list can be preceded by an integer constant indicating the number of repetitions of that variable. The integer and data values are separated by an asterisk. For example:

```
DATA A,B,C/3*3.17/
```

The three real variables are each assigned the value 3.17.

Radix-5Ø constants (See Section 7-2.2.9) may only be used to initialize variables of type REAL.

Hollerith (or character) constants may be used to initialize variables of any type. Each Hollerith constant may be used to initialize exactly one variable. If the number of characters in a Hollerith constant is fewer than needed to initialize the entire variable element, the compiler will append additional blank characters to the right of the constant to completely fill the variable. If the number of characters in the constant is more than needed, the leftmost characters will be used and the remainder ignored.

The specified values are compiled into the object program and are assumed by the variables when program execution begins. Variables in a labeled common block can only be defined in a BLOCK DATA subprogram (see Section 7-6.4.8). Variables in the blank common block cannot be initialized.

Variables may also be subscripted array elements or unsubscripted array names. When the unsubscripted name of a predefined array is given in the list, a data value must be specified for every element of the array. Data elements are stored in the array in the order used for data transmission and storage of arrays; that is, in order of increasing subscripts with the first subscript varying most rapidly (column by column storage, see Section 7-5.3.1). Allocation of data to an array stops when

a. the data (value) list is exhausted; or

b. data items have been allocated to the entire array, in which case additional data items are allocated to any additional items in the variable list.

For example,

```
DATA X,Y,Z/'A','BCDE','FGHIJKL'/
```

produces the following in memory.

X+1	X
blank	A
X+3	X+2
blank	blank
Y+1	Y
C	B
Y+3	Y+2
E	D
Z+1	Z
G	F
Z+3	Z+2
I	H

The data items following each list of variables must have a one-to-one correspondence with the variables of the list, and must agree in type, since each item of the data specifies the value given to the corresponding variable.

Data items assigned may be numeric, Hollerith, octal, hexadecimal, Radix-5Ø, or logical constants. For example,

```
DATA ALPHA,BETA,IVAL/5.,16.E-2,.TRUE./
```

specifies the value 5.Ø for ALPHA, the value Ø.16 for BETA, and the value -1 for IVAL. Any item of data may be preceded by an integer constant followed by an asterisk. This notation indicates that the item is to be repeated. Example:

```
DATA A(1),A(2),A(3)/3*Ø./
```

specifies the value Ø.Ø for array elements A(1) to A(3). For example,

```
DIMENSION A(2,2),B(3)
DATA A,B/2*1 .Ø,3*2.Ø,3.Ø,4.Ø/
```

will initialize

```
A(1,1) and A(2,1) to 1.Ø
A(1,2), A(2,2) and B(1) to 2.Ø
B(2) to 3.Ø, and B(3) to 4.Ø  .
```

As another example,

```
DIMENSION R(4),I(3),L(5)
DATA L,I,R/4*O177777,2*'AB', 2*4,4*1.Ø/
```

results in

```
L(1) to L(4) = 177777 (octal)
L(5),I(1) = 'AB' (Hollerith)
I(2),I(3) = 4
R(1) to R(4)= 1.Ø
```

PART 7

CHAPTER 6

FORTRAN FUNCTIONS AND SUBROUTINES

6.1 PROCEDURES

FORTRAN contains provisions for the creation and use of coded procedures. Procedures consist of FORTRAN (or assembly language) routines that perform operations for, and independently of, the main FORTRAN program. Procedures perform operations that may be required more than once in the course of a program and that might otherwise require repetition of code within a program. Their use, then generally saves programming effort and reduces storage used.

There are two main types of procedures: functions and subroutines. Functions include the following.

1. library functions (provided by the FORTRAN OTS)
2. statement functions (one-line, user-defined function definitions)
3. external functions (user-defined function subprograms)

Subroutines include the following.

1. system subroutines (provided by the FORTRAN OTS)
2. subroutines (user-defined subroutine subprograms)

Library functions and system subroutines are provided by the FORTRAN Object Time System and need only be referred to or called by the user program. (See Section 7-6.2 and Chapter 7-14 for a description of the available routines.) External functions and subroutines can be compiled independently of the main program; they are linked to the main program by the Linker Program (LINK).

The term subprogram refers to both external functions and subroutines. Subprograms can communicate with the main program and among themselves by means of parameters specified in the subprogram call or by means of variables stored in common blocks. Calls to subprograms cannot be recursive; that is, a call to any subprogram cannot result in a subsequent call to that subprogram, directly or indirectly.

6.2 FORTRAN LIBRARY FUNCTIONS

The FORTRAN Library functions are described in Table 7-3. In order to use a library function in any routine, it is only necessary to use the symbolic name of the function, together with one or more data references (arguments) upon which the function is to act, in a FORTRAN statement. The value obtained from the execution of the function is assigned to the function's name, which can then be used as a variable in the execution of the FORTRAN statement. For example,

```
R = ABS(X-1)
```

causes the absolute value of X-1 to be calculated and assigned to the variable R.

The data type of each library function is predefined as described in Table 7-3. Arguments passed to these functions may consist of subscripted or simple variable names, constants, arithmetic expressions or other intrinsic functions. Arguments to these functions must correspond to the type indicated in Table 7-3. If the argument types do not match, no conversion is performed, nor is any diagnostic given. See also Section 7-5.1 for the effect of the IMPLICIT statement on function values.

Details on the algorithms used in the individual library functions are contained in Chapter 7-13.

6.3 ARITHMETIC STATEMENT FUNCTIONS

An arithmetic statement function is used to define one-line functions for the user. These functions can be referenced only by the program unit in which they are defined.

To define an arithmetic statement function, the following form is used:

name(arg_1,...) = expression

where name is the function name, which must be a legal FORTRAN variable name. Function type is determined either implicitly, by the initial letter of the function name, or explicitly, through a data type specification. The term (arg_1,...) represents the list of dummy variables used in the function definition; at least one argument must be present. Expression is an arithmetic expression that defines the computation to be performed by the function.

Table 7-3

FORTRAN Library Arithmetic Functions

Form	Definition	Argument Type	Result Type
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value where Z=(X,Y) CABS(Z)=$(X^2+Y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion, obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion CMPLX(X,Y)=X+Y*i	Real	Complex
	Truncation functions return the largest integer ≤ |argument|, carrying the sign of the argument.		
AINT(X)	Real to Real truncation	Real	Real
INT(X)	Real to Integer truncation	Real	Integer
IDINT(X)	Double to Integer truncation	Double	Integer
	Remainder functions divide the first argument by the second and return the remainder from that division		
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
	Maximum value functions return the largest value in a list of at least two arguments.		
AMAXØ(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAXØ(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real List	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double

Table 7-3 (Cont.)

FORTRAN Library Arithmetic Functions

Form	Definition	Argument Type	Result Type
	Minimum value functions return the smallest value in a list of at least two arguments.		
AMINØ(I,J,...)	Real minimum from Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum from Real list	Real	Real
MINØ(I,J,...)	Integer minimum from Integer list	Integer	Integer
MIN1(X,Y,...)	Integer minimum from Real List	Real	Integer
DMIN1(X,Y,...)	Double minimum from Double list	Double	Double
	The transfer of sign Functions return the value of the first argument to which the sign of the second has been attached.		
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(X,Y)	Double precision transfer of sign	Double	Double
	Positive difference functions return the first argument minus the lesser of the two arguments (never less than zero).		
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
	Exponential functions return the value of *e* raised to the power of the argument.		
EXP(X)	e^X	Real	Real
DEXP(X)	e^X	Double	Double
CEXP(Z)	e^Z	Complex	Complex
ALOG(X)	Returns $\log_e(x)$	Real	Real
ALOG1Ø(X)	Returns $\log_{10}(X)$	Real	Real
DLOG(X)	Returns $\log_e(X)$	Double	Double
DLOG1Ø(X)	Returns $\log_{10}(X)$	Double	Double
CLOG	Returns $\log_e$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex

Table 7-3 (Cont.)

FORTRAN Library Arithmetic Functions

Form	Definition	Argument Type	Result Type

COS(X)
DCOS(X)
CCOS(Z)
	Real cosine
	Double precision cosine
	Complex cosine
		Real
		Double
		Complex
			Real
			Double
			Complex

TANH(X)	Hyperbolic tangent	Real	Real

ATAN(X)
DATAN(X)
ATAN2(X,Y)
DATAN2(X,Y)
	Real arctangent
	Double precision arctangent
	Real arctangent of (X/Y)
	Double precision arctangent of (X/Y)
		Real
		Double
		Real
		Double
			Real
			Double
			Real
			Double

CONJG(Z)	Complex conjugate, if Z=X+Y*i CONJG(Z)=X-Y*i	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range Ø to 1. I and J must be integer variables and should be set initially to Ø. Resetting I and J to Ø regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences.	Integer	Real

Arguments of the sine and cosine functions represent angles expressed in radians.

Arithmetic statement function definitions must occur after all declaratives (specification statements) and before any executable statements.

An arithmetic statement function definition must be contained ir a single statement. The expression that defines the function may include dummy arguments, other variables, array elements, non-Hollerith constants, and references to intrinsic functions, external functions, and previously-defined statement functions. For example,

```
     ACOSH(X,A) = (EXP(X/A))/2.Ø
```

is an acceptable arithmetic statement function definition. X and A are dummy arguments. Since they serve only to indicate the number, type, and order of actual arguments, they may also be used as variables elsewhere in the program unit.

Array references cannot be used as dummy arguments in a statement function definition. For example,

```
     DIMENSION X(1Ø)
     F(X) = Y+X(1)+X(2)+X(3)
```

would generate a function reference to the array X instead of an array reference. No diagnostic is issued. However, in the following definition,

```
     G(Y) = F(Y)*X(Y+2)
```

X is legal as an array name since it is not a dummy argument.

6.4 EXTERNAL SUBPROGRAMS

6.4.1 Subprogram Arguments

Variable names and identifiers used in any given program unit are completely independent of variable names in other program units. Relationship between data references in different program units (and, thereby, communication of data between those program units) is established by means of arguments. The variable names and identifiers that appear in a function reference or a subroutine CALL statement are called actual arguments; those that appear in a FUNCTION or SUBROUTINE statement that defines an external subprogram are called dummy arguments. The rules concerning dummy arguments are identical for both function and subroutine subprograms.

Subprograms are written much like a FORTRAN main program. However, the first statement of a subprogram must be either a FUNCTION statement with a list of arguments or a SUBROUTINE statement with or without an argument list. The argument list contains the dummy arguments of the subprogram. When the subprogram is called, the dummy arguments are replaced by the actual arguments contained in the calling statement.

A subroutine may return zero, one, or more values to the calling program by means of its arguments. A function returns a single value, which is assigned to the function's name. The values of a function's arguments may, however, be altered within the function to effectively return additional values to the calling program unit. Actual arguments that are passed to a subprogram may be any of the following.

arithmetic or logical expressions
alphanumeric strings

array names or elements
subprogram names

(Expressions may contain constants, variables, array elements, and library function references, alone or in conjunction with operators.)

Actual arguments must agree in number, order, and type with the dummy arguments of the subprogram to which they are passed. No type conversion is performed where a difference in data type occurs, nor is any diagnostic given.

Dummy arguments that represent array names must be defined within the subprogram, either in a DIMENSION statement or in one of the type specification statements that can contain dimensioning information. Array dimensions within a subprogram, although they need not be exactly the same as those of the corresponding array(s) in the calling program unit, must not exceed those dimensions. Dummy arguments may be used as dimension specifications within a subprogram; the actual dimensions may then be passed to the subprogram by the calling program unit (see Section 7-6.4.2).

Dummy arguments must not appear in EQUIVALENCE, DATA, or COMMON statements within a subprogram.

A function or subroutine subprogram may be used as an actual argument of a subprogram, provided that name appears in an EXTERNAL statement in the calling program unit. The EXTERNAL statement identifies such a name as being an external subprogram name, rather than a variable name.

For example, the subprograms TRIG and TAN:

```
SUBROUTINE TRIG(ARG,FUNC,RES)
RES = FUNC(ARG)
RETURN
END

FUNCTION TAN(ARG)
TAN = SIN(ARG)/COS(ARG)
RETURN
END
```

If the calling program unit contains an EXTERNAL statement naming the library functions SIN and COS, and the user-written function TAN, three different calls to the subroutine TRIG can be made to produce three different results. For example:

```
 .
 .
EXTERNAL SIN,COS,TAN
 .
 .
CALL TRIG(ANGLE,SIN,SINE)
 .
 .
 .
CALL TRIG(ANGLE,COS,COSINE)
 .
 .
 .
CALL TRIG(ANGLE,TAN,TANGNT)
```

Because each call transmits a different function name to the subroutine TRIG, the first call returns the sine of the first argument, the second call returns the cosine, and the third, the tangent.

6.4.2 Adjustable Dimensions

In order to obtain maximum flexibility from a subprogram, the user may wish to vary the dimensions of one or more arrays with each subprogram call. An array within a subprogram may have adjustable dimensions, provided it refers to an array defined in the calling program unit and passed as an argument.

In this case, both the array name and its dimension values must be expressed as dummy arguments in the FUNCTION or SUBROUTINE statement that begins the subprogram. The array name (which may be of any data type) and the dimension variables (which must be simple integer variables) are used to define the array within the subprogram as follows.

```
SUBROUTINE TRAN(A,I,J,RET,DET)
DIMENSION A(I,J)
 .
 .
 .
RETURN
END
```

The dimension arguments passed in this way must not exceed the actual dimensions of the corresponding array, as defined in the calling program unit, although they may be smaller.

The arguments representing the array dimensions must be defined prior to the subprogram call; the dimension values will not change during the execution of the subprogram even though the dummy arguments may be redefined or made undefined within the subprogram. The dimension specifications for an adjustable array must be supplied as arguments within the subprogram call; they cannot reside in common.

6.4.3 FUNCTION Subprograms

A FUNCTION subprogram is a computational procedure that returns a single value through its name; a function can also return values through its arguments. A FUNCTION subprogram is invoked by using the function name with associated arguments in an arithmetic expression. A FUNCTION subprogram begins with a FUNCTION statement and ends with an END statement. Control is returned to the calling statement by means of one or more RETURN statements. (The END statement acts like an implied RETURN statement.) The mode of the function (the type of value returned) is determined by the name of the function or a type designation inserted into the initial FUNCTION statement.

The FUNCTION statement format:

FUNCTION name(arg_1,...arg_n)

or

type FUNCTION name(arg_1,...,arg_n)

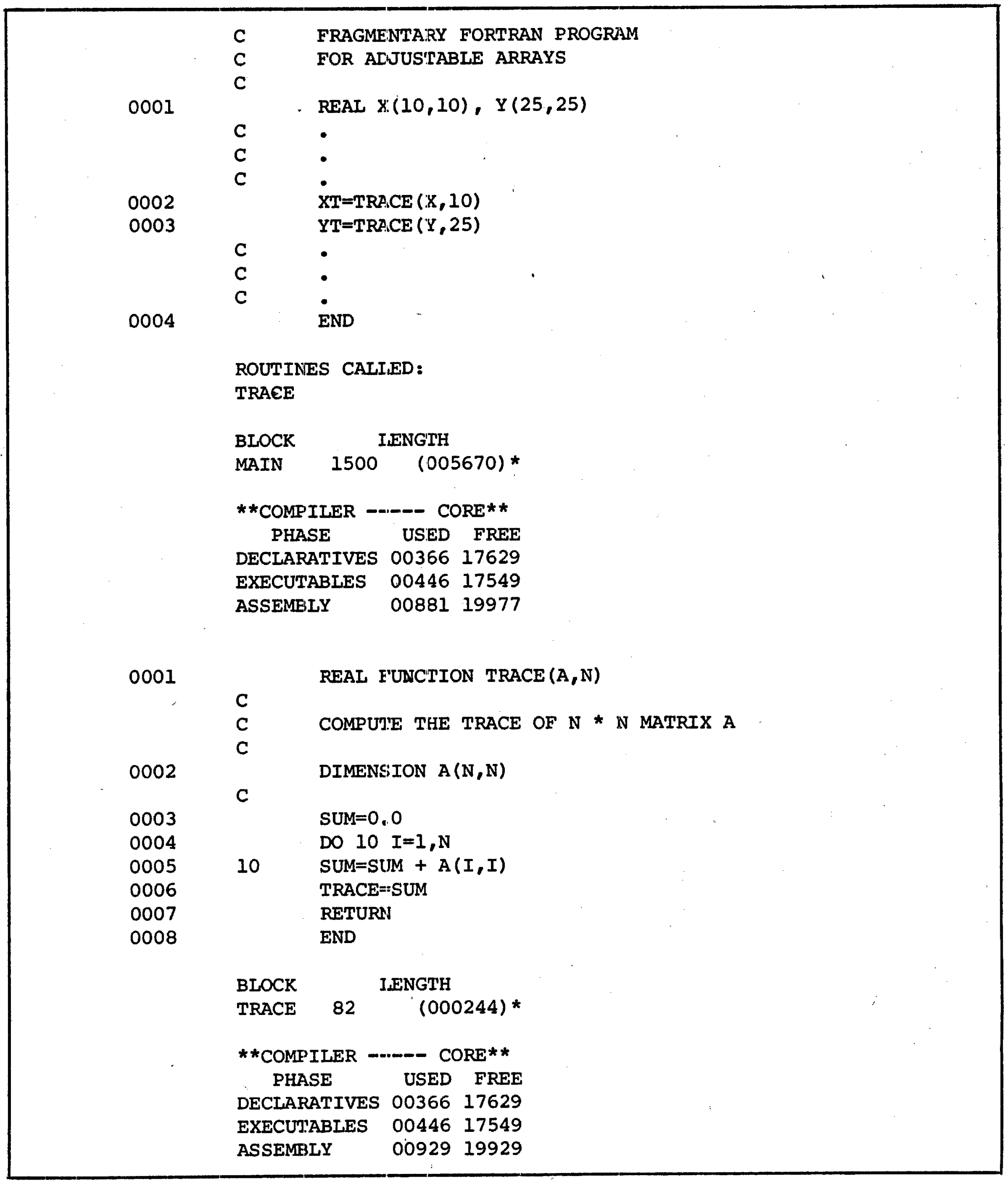
This statement declares the program unit that follows to be a FUNCTION subprogram. The function name may be any legal FORTRAN variable name that is not used as a dummy argument. This name must not appear in any nonexecutable statement in the calling program or function subprogram, except as a scalar variable in a type declaration. At some point in the subprogram the function name must appear as a scalar variable to which a value is given by an assignment statement. The value assigned to the function name is the value returned by the function reference in the calling program.

The argument list in the FUNCTION statement must consist of at least one dummy argument. Dummy arguments follow the rules established for subprogram parameters as described in Section 7-6.4.1 and 7-6.4.2.

The data type of the value returned by the function is normally determined to be either INTEGER or REAL according to the default type rule applied to the function name. However, a different type can be defined for the returned value by placing a type designation term at the beginning of the FUNCTION statement or placing a type declaration statement within the subprogram.

A FUNCTION statement must not have a statement number. The only FORTRAN statements that must not appear in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA and another FUNCTION statement.

An example of a FUNCTION subprogram is shown in Figure 7-3.



```
                 C      FRAGMENTARY FORTRAN PROGRAM
                 C      FOR ADJUSTABLE ARRAYS
                 C
      0001              REAL X(10,10), Y(25,25)
                 C      .
                 C      .
                 C      .
      0002              XT=TRACE(X,10)
      0003              YT=TRACE(Y,25)
                 C      .
                 C      .
                 C      .
      0004              END

                 ROUTINES CALLED:
                 TRACE

                 BLOCK        LENGTH
                 MAIN    1500    (005670)*

                 **COMPILER ----- CORE**
                    PHASE       USED  FREE
                 DECLARATIVES 00366 17629
                 EXECUTABLES  00446 17549
                 ASSEMBLY     00881 19977


      0001              REAL FUNCTION TRACE(A,N)
                 C
                 C      COMPUTE THE TRACE OF N * N MATRIX A
                 C
      0002              DIMENSION A(N,N)
                 C
      0003              SUM=0.0
      0004              DO 10 I=1,N
      0005       10     SUM=SUM + A(I,I)
      0006              TRACE=SUM
      0007              RETURN
      0008              END

                 BLOCK        LENGTH
                 TRACE   82      (000244)*

                 **COMPILER ----- CORE**
                    PHASE       USED  FREE
                 DECLARATIVES 00366 17629
                 EXECUTABLES  00446 17549
                 ASSEMBLY     00929 19929
```

Figure 7-3

Sample FUNCTION Subprogram

6.4.4 SUBROUTINE Subprograms

A SUBROUTINE subprogram is a computational procedure that returns zero, one, or more values to the calling program. Values and data types are associated with the subroutine arguments, but not with the subroutine name. A SUBROUTINE subprogram begins with a SUBROUTINE statement and ends with an END statement. Control is returned to the calling statement by means of one or more RETURN statements.

The SUBROUTINE statement is of the following form.

    SUBROUTINE name

or

SUBROUTINE name(arg$_1$,...,arg$_n$)

This statement declares the program that follows to be a SUBROUTINE subprogram. The subroutine name may be any legal FORTRAN variable name, but must not be used as a variable name within the calling program or subprogram itself, or as the name of a labeled COMMON block used in the same load module.

The argument list in the SUBROUTINE statement consists of any number of dummy arguments (or none). Dummy arguments follow the rules established for subprogram parameters as described in Section 7-6.4.1 and 7-6.4.2. Dummy arguments to a SUBROUTINE subprogram must not appear in an EQUIVALENCE or COMMON statement within the subprogram.

A SUBROUTINE subprogram can redefine any of the actual arguments or use any of the dummy arguments specified to return results to the calling program. For example:

```
      SUBROUTINE COMPUT (A,B,ANS)
      ANS=A+B
      A=A+1.Ø
      RETURN
      END
```

In this simple subroutine, the calling statement provides numeric values for A and B and a variable name in which the value of ANS is returned. If the calling statement is

```
      CALL COMPUT (AI,4.Ø,RET)
```

where the value of AI is 7.Ø, the value returned to the calling program in RET is 11.Ø, and AI is 8.Ø.

A SUBROUTINE subprogram cannot contain a FUNCTION, BLOCK DATA or another SUBROUTINE statement.

6.4.5 CALL Statement

A subroutine is invoked by means of a CALL statement having one of the following forms:

```
      CALL name
```

or

```
      CALL name(arg1,...argn)
```

where name is the name of the subroutine referenced. The argument list of the subroutine call, where included, should agree in number, order, and type with the arguments in the subroutine definition. The actual arguments specified in the subroutine call replace the dummy arguments during execution of the subroutine.

The arguments of a CALL statement may be expressions, array names or elements, scalar variables, alphanumeric strings, or subprogram names. The subroutine name cannot be referenced as a basic element in an expression in the calling program or in the subroutine itself.

A call to a subroutine may contain fewer, or more, actual arguments that the subroutine definition, as long as

a. an unsupplied call argument is not a dummy array in the subroutine, and

b. some indication of the number of real arguments (such as an argument that states the number of other arguments supplied on a given subroutine call) is given to the subroutine so that no attempt is made to reference unsupplied arguments.

6.4.6 RETURN Statement

The RETURN statement is of the form

```
      RETURN
```

and causes an exit from the subprogram to the calling program. It is generally the last executable statement in the subprogram. Any number of RETURN statements can appear in either a FUNCTION or SUBROUTINE program.

The RETURN statement must not occur in a main program.

En END statement must occur following each subprogram. If a subprogram contains an END statement and no RETURN statement, executing the END statement causes a return to the calling program.

6.4.7 BLOCK DATA Subprograms

The BLOCK DATA subprogram is used to enter initial values for variables assigned to labeled COMMON blocks. The BLOCK DATA subprogram is of the following form.

```
      BLOCK DATA
        .
        .     (Specification Statements)
        .
      END
```

The first statement must be a BLOCK DATA statement, which does not allow either a line number or any arguments. No executable statements may appear in a BLOCK DATA subprogram. The entire BLOCK DATA subprogram can contain only EQUIVALENCE, DATA, DIMENSION, COMMON, IMPLICIT and type declaration statements. A complete set of specifications is given for an entire common block by a single BLOCK DATA subprogram, although a single BLOCK DATA subprogram can initialize any number of named common blocks. For example:

```
      BLOCK DATA
      COMMON/R/S,Y/C/Z,W,Y
      DIMENSION Y(3)
      COMPLEX Z
      DATA Y/1E-1,2*3E2/W,Z/11.877EØ,(-1.41421,1.41421)/
      END
```

6.5 NULL ARGUMENTS IN CALLS TO SUBROUTINES OR FUNCTIONS

An argument in a call to a subroutine or function may be left unspecified by writing a null, or blank, expression as the actual argument. For example,

```
      CALL S(A,,B)
```

is a call with three actual arguments of which the second argument is null.

The use of null arguments is only meaningful when calling routines that are specifically designed to accept optional arguments. Such routines can only be written in assembly language, not in the PDP-11 FORTRAN language.

In general, the use of a null argument in calling a routine that is not designed to detect and handle it will result in a fatal error at execution time.

The implementation of null arguments is discussed in Chapter 7-15.

PART 7

CHAPTER 7

FORMAT STATEMENTS

Data transmission statements govern the transfer of data between internal storage and either peripheral devices or other locations in the computer memory. These statements are divided into four categories.

a.	Data description statements:	Format (described in this Chapter) and DEFINE FILE (Section 7-8.4.1)
b.	Input/Output statements	READ, WRITE and PRINT (Chapter 7-8)
c.	Device control statements for magnetic tape and disk usage:	FIND, BACKSPACE, REWIND and END FILE Section 7-8.4.2, 7-8.6.1, 7-8.6.2, and 7-8.6.3)
d.	Statements specifying data transmission between internal storage areas:	ENCODE and DECODE (Section 7-8.7)

Chapter 7-7 describes the construction of FORMAT statements. Chapter 7-8 describes the remaining data transmission statements.

7.1 FORMAT STATEMENTS

The FORMAT statement describes the format in which one or more records are transmitted. FORMAT statements are nonexecutable statements used in conjunction with input/output statements or with ENCODE and DECODE statements.

FORMAT statements have the following form.

FORMAT(S_1,S_2,...S_n /S'_1, S'_2,....S_n/...)

A statement number must be specified. (FORMAT statements are referenced by the data transmission statements with which they operate.) Each S is a data field specification; the slash (/) character indicates the start of a new record. The closing parenthesis at the end of the FORMAT statement also terminates a record.

Each record description may consist of one or more field specifications, a field being a consecutive series of characters within the record. Field specifications are separated by commas within the FORMAT statement (consecutive commas are not allowed).

FORMAT statements must be placed in the executable part of the source program. Unless the FORMAT statement contains only alphanumeric data for direct transmission, it is used in conjunction with the list of a data transmission statement. The FORMAT statement indicates the form in which the variables specified in the data transmission statement are to be read or written.

During data transmission, the object program scans the indicated FORMAT statement. If one or more specifications for a numeric field, for example, are present, and the data transmission statement contains numeric data items to be transmitted, those items are transmitted according to the format specification. Execution of the data transmission statement is terminated when all items in the data list have been handled. The FORMAT statement can contain specifications for more items than are contained in a given data transmission statement, in which case excess format specifications are ignored. Conversely, the FORMAT statement can contain specifications for fewer items than are contained in the data transmission list, in which case the set of format specifications beginning with the last left parenthesis or slash is repeated as often as necessary to supply the needed number of data formats.

A field specification within a FORMAT statement is of the form

Cw

or

Cw.d

where C is a format code, w indicates the field width and d (where present) indicates the number of characters within the field that occur following the decimal point. The following types of format conversion codes may appear in a FORMAT statement.

a.	numeric:	Iw, Ow, Fw.d, Ew.d, Dw.d, Gw.d
b.	numeric with scale factor:	nPFw.d, nPEw.d, nPDw.d, nPGw.d
c.	logical:	Lw
d.	alphanumeric and editing	Aw, nH, '...', nX, Tw

There are several ways to specify the numeric values for repeat count, (see Section 7-7.19) field width and number of characters following the decimal point in formats. They are used in the following order of precedence.

1. A default value is assumed. The default repeat count is 1. The default values for width and number of decimals are discussed in Section 7-7.14.
2. If an explicit value is specified, it is used in place of the default. Explicit values may be specified in the format either as a literal integer or as a variable format expression (Section 7-7.17).
3. In the case of formatted input, if the field thus specified contains a comma or extends past the end of the input record, then the field is said to be short field terminated and the field width (but not the number of decimal characters) is reduced accordingly. See Section 7-7.15.

7.2 I FORMAT CONVERSIONS

A specification of the form

Iw

is used to transmit decimal integer values. The corresponding element in the data transmission list must be an integer or byte variable.

On output, a field w characters long is reserved for the value to be output. The value is transmitted into this field right-justified. Where a minus sign is expected in the output, one position must be included in w for that sign. Since a plus sign is suppressed, no allowance need be made for it. Where the field specified for the output value is too small, the entire output field is filled with asterisks. For example:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	ΔΔ174
I2	3244	**
I3	-473	***
I7	29.443	not permitted, error

On input, a field of w positions is read as it appears. The external data must be a decimal integer and must not contain a decimal point or exponent field. Blanks are interpreted as zeroes. Leading blanks are ignored. An all blank field is interpreted as a zero on input. An integer overflow occurs if the value being read is beyond the range -32768 to 32767. For example:

Format	External Value	Internal Representation
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	312
I9	312ΔΔΔΔΔΔ	not permitted, error; system attempts to input 312ØØØØØØ

7.3 O FORMAT CONVERSIONS

A specification of the form

Ow

is used to transmit octal integer values. The corresponding element in the data transmission list must be an integer or byte variable.

On output a field w characters long is reserved for the value. The value is transmitted into this field right-justified. Where the field specified is too small, the entire output field is filled with asterisks. For example:

Format	Internal (Decimal)	External Representation
O6	32767	Δ77777
O6	-32768	1ØØØØØ
O2	14261	**
O4	27	ΔΔ33
O5	13.52	not permitted, error

On input, a field of w positions is read as it appears. The external values must not contain a sign, a decimal point, an exponent field, or the digits 8 or 9. Leading blanks are ignored, embedded and trailing blanks are interpreted as zeros; an all blank input field is treated as a value of Ø. The value is read into the first word of the data element specified in the input statement. For example:

Format	External Value	Internal Octal Representations
O5	32767	32767
O4	16234	1623
O6	13ΔΔΔΔ	13ØØØØ
O3	97	not permitted, error

7.4 F FORMAT CONVERSIONS

A specification of the form

Fw.d

is used to transmit real or complex values. The corresponding element in the data transmission list must be a real or complex variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The total field length reserved (w) must be large enough to include a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), and d decimal digits following the decimal point; that is, w≥d+3. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For F conversion output, the data value after scaling must be within the range 0.14×10^{-38} to 1.7×10^{38}. For example:

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.3	51.44	**
F1Ø.4	-23.24352	ΔΔ-23.2435
F5.2	325.Ø13	*****
F5.2	-.2	-Ø.2Ø

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. Blanks are interpreted as zeros. Leading blanks are ignored. An all blank field is interpreted as zero. Where no decimal point is encountered within the first w positions, those digits are accepted as the input value and a decimal point is inserted d positions from the right. A decimal point found in the input value overrides the format specification. For example:

Format	External Value	Internal Representation
F8.5	123456789	123.45678
F8.5	12345.67	12345.67
F8.5	24.77E+2	2477.Ø
F5.2	1234567.88	123.45

7.5 E FORMAT CONVERSIONS

A specification of the form

Ew.d

is used to transmit real or complex values, with a decimal point and E exponent field. The corresponding element in the data transmission list must be a real or complex variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The total field length reserved (w) must be large enough to include a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), d decimal digits following the decimal point, and four positions for the exponent representation; that is to say, $w \geq d+7$ in most cases. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For E conversion output, the data value after scaling must be within the range 0.14×10^{38} to 1.7×10^{38}. For example:

Format	Internal Value	External Representation
E9.2	475867.222	ΔØ.48EΔØ6
E12.3	475867.222	ΔΔΔØ.476EΔØ6
E5.3	56.12	*****

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without decimal point or exponent field. Blanks are interpreted as zeros. Leading blanks are ignored. An all blank field is interpreted as zero. Where no decimal point is encountered within the first w positions, those digits are accepted as the input value and a decimal point is inserted d positions from the right; a zero exponent is assumed. A decimal point found in the input value overrides the format specification. For example:

Format	External Value	Internal Representation
E9.3	734.432E8	734.432E8
E12.4	ΔΔ1Ø22.43E-6	1Ø22.43E-6
E15.3	52.3759663	52.3759663
E1Ø.4	Δ1971.3213E12	1971.3213
E8.1	123456789	1234567.8EØ
E12.5	21Ø.5271D+1Ø	21Ø.5271E1Ø

7.6 D FORMAT CONVERSIONS

A specification of the form

Dw.d

is used to transmit real values, with a decimal point and D exponent field. The corresponding element in the data transmission must be a real or double-precision variable.

On output, the effect of this field specification is identical to that of an equivalent Ew.d specification, except that a D exponent indicator is used in place of the E indicator. For example:

Format	Internal Value	External Representation
D.14.3	Ø.363	ΔΔΔΔΔØ.363DΔØØ
D23.12	5413.87625793	ΔΔΔΔΔØ.541387625793DΔØ4
D9.6	1.3	*********

On input, the effect of a Dw.d specification is identical to an equivalent Ew.d specification, except that the internal representation appears as in the following examples.

Format	External Value	Internal Representation
D10.2	12345ΔΔΔΔΔ	12345ØØØ.ØDØ
D10.2	ΔΔ123.45ΔΔΔ	123.45DØ
D15.3	367.4981763D+Ø4	3.674981763D+Ø6

7.7 G FORMAT CONVERSIONS

A specification of the form

Gw.d

is used to transmit real or complex data.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The form of the output conversion depends upon the individual variable in the data transmission list. The output conversion format is a function of the magnitude of the data being converted, as described in Table 7-4.

Table 7-4

Floating-Point Magnitudes and Resulting G Format Conversions

Data Magnitude	Effective Conversion
m < Ø.1	Ew.d
Ø.1 ≤ m < 1.Ø	F(w-4).d, 4X
1.Ø ≤ m < 1Ø.Ø	F(w-4).(d-1), 4X
.	.
.	.
.	.
$10^{d-2} \leq m < 10^{d-1}$	F(w-4).1, 4X
$10^{d-1} \leq m < 10^{d}$	F(w-4).Ø, 4X
$m > 10^{d}$	Ew.d

(The 4X specification, which is inserted (in effect), by the Gw.d specification, indicates that four blanks are to follow the numeric field representation.. The X specification is described in Section 7-7.16).

The total field length reserved (w) includes a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), d decimal digits following the decimal point, and four positions for the exponent representation; that is to say, $w \geq d+7$ in most cases. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For example:

Format	Internal Value	External Representation
G13.6	Ø.Ø1234567	ΔØ.123457E-Ø1
G13.6	Ø.12345678	ΔØ.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ
G13.6	12.3456789Ø	ΔΔ12.3457ΔΔΔΔ
G13.6	123.456789ØØ	ΔΔ123.457ΔΔΔΔ
G13.6	1234.56789ØØØ	ΔΔ1234.57ΔΔΔΔ
G13.6	12345.6789ØØØØ	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.789ØØØØØ	ΔΔ123457.ΔΔΔΔ
G13.6	1234567.89ØØØØØØ	ΔØ.123457EΔØ7

For comparison, consider the following example of the same values output under the control of an equivalent Fw.d specification.

Format	Internal Value	External Representation
F13.6	Ø.Ø1234567	ΔΔΔΔΔØ.Ø12346
F13.6	Ø.12345678	ΔΔΔΔΔØ.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.3456789Ø	ΔΔΔΔ12.345679
F13.6	123.456789ØØ	ΔΔΔ123.456789
F13.6	1234.56789ØØØ	ΔΔ1234.56789Ø
F13.6	12345.6789ØØØØ	Δ12345.6789ØØ
F13.6	123456.789ØØØØØ	123456.789ØØØ
F13.6	1234567.89ØØØØØØ	*************

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. The rules for input conversion are the same as those for F format input conversion.

7.8 COMPLEX I/O

Complex data values are transmitted as two independent real qualities. The following specification consists of one repeated real specification or two successive real specifications of the form.

Two Real Specifications	Repeated Real Specifications
Fw.d,Fw.d	2Fw.d
Ew.d,Ew.d	2Ew.d
Gw.d,Gw.d	2Gw.d

Nothing is output between the two parts of a complex number unless specified by the format.

Output Conversion:

Format	Internal Value	External Representation
2F8.5	2.3547188,3.456732	Δ2.35472Δ3.45673
E9.2,E5.3	47587.222,56.123	ΔØ.48EΔØ6*****

Input Conversion:

Format	External Value	Internal Representation
F8.5,F8.5	1234567812345.67	123.45678,12345.67
E9.1,E9.3	734.432E8123456789	734.432E8,1234567.8E-1

7.9 SCALE FACTORS

Scale factors of the form

nP

can be specified for D, E, F, and G conversions. The number n is a signed or unsigned integer specifying the scale factor.

For E and D type output conversions, the basic real constant part of the output quantity is multiplied by 1Ø(n) and the exponent is reduced by n. For G type output conversion, the effect of the scale factor is suspended unless the magnitude of the value to be converted is outside the F conversion range (see Table 7-4).

For the F, E, G and D type input conversions where no exponent is found in the external value, and F type output conversions, the scale factor causes the external value to equal the internal value times 1Ø (n).

For F, E, G and D type input conversions, the scale factor has no effect if there is an exponent in the external value.

Where no scale factor is specified, a scale factor of zero is assumed. Once a scale factor has been specified, it is carried onto all subsequent D, E, F, and G type conversions within the same format specification unless another scale factor is encountered. A zero scale factor can only be reinstated by an explicit ØP specification.

Output Conversions:

Format	External Value	External Representation
3PE15.3	-273.612	ΔΔΔ-273.612EΔØØ
1PE15.3	-273.612	ΔΔΔΔΔ-2.736EΔØ2
1PE15.2	-273.612	ΔΔΔΔΔΔ-2.74EΔØ2
-1PE15.2	-273.612	ΔΔΔΔΔΔ-Ø.Ø3EΔØ4

Input Conversions:

Format	External Value	Internal Representation
3PE1Ø.5	ΔΔΔ37.614	.Ø37614
3PE1Ø.5	ΔΔ37.614E2	3761.4
-3PE1Ø.5	ΔΔΔΔΔ37.614	37614.Ø

7.10 L FORMAT CONVERSIONS

A specification of the form

Lw

is used to transmit the value of a logical variable.

On output, a field w characters long is reserved for the value, which is transmitted into this fielld right-justified. The letter T or F is transmitted following w-l blanks. For example:

Format	Internal Value	External Representation
L5	.TRUE.	ΔΔΔΔT
L1	.FALSE.	F

On input, the first nonblank character in the data field must be a T or an F; the value of the logical variable is stored as true or false, respectively. If the data field is blank or empty, a value of false is stored. Any other value in the data field causes an error diagnostic.

7.11 A FORMAT CONVERSIONS

Alphanumeric data can be transmitted in a manner similar to numeric data by use of a specification of the following form.

Aw

Conversions of this type are generally used where the alphanumeric data is generated or operated upon by the user program.

The field specified has a length of w characters. The alphanumeric characters are transmitted as the value of a variable in the data transmission list. The variable may be of any type. The value of w is limited to the maximum number of characters that can be stored in the space allotted for a single variable of the type specified. Table 7-5 indicates the number of alphanumeric characters that can be stored in a single variable of a given type.

Table 7-5

Alphanumeric Data Storage

Variable Type	Number of Characters per Variable
Integer	2
Byte	1
Logical	2
Real	4
Double-precision	8
Complex	8

For example, the sequence

```
      READ(2,5) V
    5 FORMAT (A4)
```

causes four characters to be read and stored in memory as the value of the real variable V.

If the value of w exceeds the amount specified in Table 7-5, the leftmost characters are lost on input; on output, w characters appear right-justified in the external output field.

If the value of w is less than the number of characters that can be stored in the space allotted to a variable, on input the characters are left-justified and blank-filled on the right of each list item. On output the leftmost w characters in the variable are transmitted to the output field.

7.12 ALPHANUMERIC DATA WITHIN FORMAT SPECIFICATIONS

Alphanumeric data can be transmitted directly into or from a format specification by means of Hollerith conversion (H) or of single quote delimiting characters. Direct transmission of alphanumeric data is performed where such data is not operated upon by the program; i.e., headings and explanations to be printed along with output data.

The H conversion is indicated by a specification of the form

    nH

where n indicates the number of characters to be transmitted. On output, the following n characters are sent to the output device. On input, n characters are read and inserted into the Hollerith conversion field of the format specification. For example:

```
C       OUTPUT HEADING TO LINE PRINTER:
        WRITE (5,1Ø) N
1Ø      FORMAT(15H OUTPUT HEADING,I3)
C       INPUT HEADING FROM KEYBOARD FOR LATER USE; UP TO 3Ø CHARS:
        READ (6,20)
2Ø      FORMAT(30H                              )
```

In line 1Ø, the leading space in the line to be output is the carriage control character (see Section 7-7.14). The 14-character heading is output, followed by the integer value of N printed in a field of three characters. The comma following the Hollerith field is optional.

In line 20, the Hollerith field indicated is blank; however, any alphanumeric characters within this field would be replaced by those characters read from the keyboard. If fewer than n characters are read, they overlay the first portion of the Hollerith field , with the remainder of the field being blank filled.

An alternative form is to enclose Hollerith data in single quotes (apostrophes). The result on both input and output is exactly as described for H conversion. An apostrophe within such a Hollerith string must be represented by two successive single quotes to distinguish it from the terminating single quote of the string. For example:

```
FORMAT('DON''T TYPE ON KEYBOARD UNTIL OUTPUT IS FINISHED')
```

When output is to the keyboard or line printer, the line

```
DON'T TYPE ON KEYBOARD UNTIL OUTPUT IS FINISHED
```

is printed.

Note that variable format expressions (Section 7-7.17) cannot be used with H conversion.

7.13 Q FORMAT SPECIFICATION

The Q format specification is used to read the actual number of characters input on the current READ operation. It must correspond to an input list item of integer type into which the character count will be placed. For example,

```
      READ(4,10)X,Y,Z,K
 10   FORMAT(3F13.5,Q)
```

reads input values into the real variables X, Y, and Z, and places the number of characters read in the integer variable K.

Q format is ignored on output.

7.14 DEFAULT FIELD SPECIFICATIONS

The w.d may be omitted from the numeric and logical conversions, in which case the following default values are used.

Format	Default
E	15.7
F	15.7
G	15.7
D	25.16
I	7
O	7
L	2
A	8

For example, the format

```
FORMAT(I,2E,A)
```

is equivalent to the format

```
FORMAT(I7,2E15.7,A8)
```

See also the discussion on short field termination (Section 7-7.15).

Note that in the case of E, F, G and D formats, the entire w.d portion of the format is defaulted; partial defaulting is not allowed.

Legal defaults: E, F
Illegal defaults: E1Ø, F.2, F.

7.15 SHORT FIELD TERMINATION ON FORMATTED INPUT

When using formatted input, either a comma in an input field or the end of the input record will override the field width specified by the FORMAT statement. This provides a simple method of free form input that is particularly useful when entering data from a keyboard. Short field termination may be used with E, F, G, I, L, and O formats. For example,

```
          READ(6,1ØØ)I,J,A,B
    1ØØ   FORMAT(2I6,2F1Ø.2)
```

and input data record

```
       1,-2,1.Ø,35
```

will result in the following assignments being performed by the READ statement.

```
       I = 1
       J = -2
       A = 1.Ø
       B = Ø.35
```

Note that the comma must be included within the field width specified by the FORMAT statement. The only effect of the comma is to adjust the field width used by the conversion routines. A zero length field is allowed. In the case of a null (zero length field) the value read will be Ø. for D, E, F or G formats, Ø and I or O formats and .FALSE. for L format.

A comma may not be used to terminate a field being read with A format. However, if the end of record occurs within the A field width, the available characters will be transmitted successfully with trailing blanks appended as necessary to fill the data element.

7.16 RECORD LAYOUT SPECIFICATIONS

Spacing control within a record is performed through the use of the X specification to generate a given number of blanks or skip a given number of characters, or the T specification which allows the user to specify "tabulation" to a given position within a record.

A specification of the form

```
       wX
```

provides for the transmission of w spaces on output or the skipping of w characters on input. The value of w must be greater than zero, For example:

```
        WRITE(5,75) I,X
  75    FORMAT(5H STEPI5,1ØX2HY=F7.3).
```

could be used to output the folloiwng line (Δ indicates a blank).

```
  STEPΔΔΔ28ΔΔΔΔΔΔΔΔΔΔY=b-3.872
```

The first blank would not be printed on the terminal or line printer as it would be considered the carriage control character. Notice that the comma following an X specification is optional.

A specification of the form

```
  Tw
```

specifies the character position in a FORTRAN record where the data transfer is to begin. The value of w must be an unsigned integer constant and indicates the (w)th character of the record on input and on nonprinted output, and the (w-1)th character of a printed output record (where the first character is a carriage control character). For example,

```
  2     FORMAT (T5Ø,'BLACK/T3Ø, 'WHITE')
```

would cause the following line to be printed.

```
  Print Position 29                       Print Position 49
     ↓                                        ↓
     WHITE                                    BLACK
```

The statements

```
  1     FORMAT (T35,"MONTH')
        READ (3,1)
```

causes the first 34 characters of the input data to be skipped, and the next five characters would replace the characters M, O, N, T, and H in storage. If an input record containing

```
  ABCb̸b̸b̸XYZ
```

is read with the format specification

```
  1Ø    FORMAT (T7,A3,T1,A3)
```

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

7.17 VARIABLE FORMAT EXPRESSIONS

An integer valued expression may be used in a FORMAT statement any place an integer may be used (except as a Hollerith count) by enclosing it in angle brackets, <expression>. For example,

```
  FORMAT (I<J+1>)
```

will cause an I conversion with a field width one greater than the value of J at the time the format is scanned. The expression will be re-evaluated each time it is encountered in the normal format scan. If the expression is not of type integer it will be automatically converted to integer prior to use. Any valid FORTRAN expression may be used including function calls and references to dummy parameters.

A complete example is shown in Figure 7-4.

The value of a variable format expression must be valid for its use in the format or an error will occur at runtime.

The FORTRAN program

```
     DIMENSION A(5)
     DATA A/1.,2.,3.,4.,5./
C
     DO 10 I = 1,10
     WRITE (5,100) I
100  FORMAT(I<MAXO(I,5)>)
10   CONTINUE
C
     DO 20 I = 1,5
     WRITE (5,101) (A(J),J=1,I)
101  FORMAT (<I>F10.<I-1>)
20   CONTINUE
     END
```

produces the following output when executed.

```
        1
        2
        3
        4
        5
         6
          7
           8
            9
             10
             1.
             2.0        2.0
            3.00       3.00       3.00
           4.000      4.000      4.000      4.000
          5.0000     5.00000    5.0000     5.0000      5.0000
```

Figure 7-4
Variable Format Expression Example

7.18 CARRIAGE CONTROL

The first character of each ASCII record controls the vertical spacing of the line printer and terminal. When an output line (record) is generated, it is constructed within the system exactly as described by the combined WRITE and FORMAT statement designations. Before output to either of the printing devices, the first character of the record is interpreted as the carriage control character and is deleted from the output line.

The carriage control characters are specified in Table 7-6

Table 7-6

Carriage Control Characters

Character	Effect
Δ blank	carriage return/line feed (advance printing position to beginning of next line)
Ø zero	carriage return/line feed/line feed (advance printing position to beginning of the line after the next)
1 one	carriage return/form feed (advance printing position to the beginning of the next line and perform a form feed where the hardware capability exists)
+ plus	carriage return (advance printing position to the beginning of the current line, allows current line to be overprinted)
$ dollar	inhibit carriage return/line feed (do not advance printing position, used when creating output lines to interact with the keyboard)

Any character other than those described in Table 7-6 that appears first in the line is deleted from the output line and treated as though it were a blank carriage control character.

7.19 REPETITION OF FIELDS, GROUPS, AND MULTIPLE RECORDS

A field specification can be repeated by preceding that specification with an unsigned integer constant denoting the number of repetitions. For example,

```
        2Ø      FORMAT(2E12.4,3I5)
```

is equivalent to

```
        2Ø      FORMAT(E12.4,E12.4,I5,I5,I5)
```

Similarly, a group of field specifications can be repeated by enclosing the group in parentheses and preceding the whole with an unsigned integer denoting the number of repetitions. For example,

```
        3Ø      FORMAT(2I8,2(E15.5,2F8.3))
```

is equivalent to

```
        3Ø      FORMAT(2I8,E15.5,F8.3,F8.3,E15.5,F8.3,F8.3)
```

Where several input/output records are specified in a single format specification, and those different records have different field specifications, a slash is used to indicate a new record. For example,

```
                WRITE(6,4Ø)K,L,M,N,O,P
        4Ø      FORMAT(3O8/I6,2F8.4)
```

is equivalent to the following.

```
                WRITE(6,4Ø)K,L,M
                WRITE(6,4Ø)N,O,P
        4Ø      FORMAT(3O8)
        41      FORMAT(I6,2F8.4)
```

A single record may be up to 133 characters long; this provides for a 132-character line printer plus a carriage control character.

The separating comma can be omitted when a slash is used. The presence of n slashes at the beginning or end of a FORMAT statement causes n blank records to be output or to be skipped on input. When n slashes appear between the first and last FORMAT specifications, (n-1) blank records are output or skipped on input.

7.20 FORMATS STORED AS DATA

The ASCII character string that makes up a format specification may be stored as the values of an array. Input/output statements may refer to the format by giving the array name instead of the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement, except for the word FORMAT and the statement number. The enclosing parentheses are included, however.

As an example, consider the following sequence.

```
              DIMENSION SKEL(2)
              READ(4,1)SKEL
       1      FORMAT(2A4)
              READ(5,SKEL)K,X
```

The first READ statement enters the ASCII string into the array SKEL. In the second READ statement, SKEL is referenced as the format-governing conversion of K and X.

As another example, the format created by the DATA statement below

```
              REAL FORM1 (3)
              DATA FORM1/'(I6','//I6',',I3)'/
              WRITE(6,FORM1) I,J,K
```

is equivalent to

```
              WRITE(6,5Ø) I,J,K
       5Ø     FORMAT(I6 //I6,I3)
```

One caution should be noted with respect to the storing of formats within arrays. Such storage does not result in the correct linkage of the I/O conversion routines from the OTS Library. For example:

```
              REAL FORM1 (2)
              INTEGER II
              LOGICAL LL
              DATA FORM1 / '(I5,','L5)'/
              II=Ø
              LL=.TRUE.
              WRITE(6,FORM1) II,LL
              END
```

If the preceding program is run, it produces the following run-time error message.

```
       FORTØØ8ØØØ    LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)
```

When this error occurs, execution of the program is terminated.

To correct this error, the programmer should include within his FORTRAN program a FORMAT statement that contains any format conversions that will be needed at run-time. This FORMAT statement need not be used or referenced by any READ or WRITE statement; it need only be present. In the preceding example, the following statement could be included immediately before the END statement.

```
      9999     FORMAT(I1,L1)
```

Also note that the variable format expression notation cannot occur in formats stored as data.

PART 7

CHAPTER 8

I/O AND DEVICE HANDLING STATEMENTS

8.1 INTRODUCTION TO FORTRAN I/O

Input of data to a FORTRAN program is performed by the READ statement. Output from a FORTRAN program is performed by either the WRITE or PRINT statement. These statements are generally used in conjunction with a FORMAT statement or format specification contained in an array. Each READ or WRITE statement contains a reference to the device to or from which the data transfer is to occur. The PRINT statement assumes output to the line printer, although the assumed device can be altered.

There are three categories of READ and WRITE statements.

a. Formatted READ, WRITE and PRINT statements are used with or without an I/O list. Where a list of variables is provided, those values are transferred and automatically converted to ASCII on output or internal format on input. Where an I/O list is not provided, an alphanumeric data transfer is indicated.

b. Unformatted READ or WRITE statements perform a transfer of binary information only. These statements make no reference to FORMAT statements or specifications.

c. Direct access disk READ and WRITE statements perform I/O of fixed-length records to a file previously defined by a DEFINE FILE statement. Direct access READ and WRITE statements cannot be used with FORMAT statements or specifications.

A feature available with all forms of READ and WRITE statements allows the program to transfer control to a specified line number if an end-of-file or error condition develops during the I/O operation.

Table 7-7 shows the skeletal formats for each of the three categories described above.

The remaining statements covered in this chapter perform file-oriented or device-oriented operations with respect to bulk storage devices such as magnetic tape, disk and DECtape.

Table 7-7

READ and WRITE Statement Summary

Category	Input Statements	Output Statements
formatted	READ (u,f) list READ (u,f)	WRITE (u,f) list WRITE (u,f)
	READ f, list	PRINT f, list
unformatted	READ (u) list READ (u)	WRITE (u) list WRITE (u)
direct disk access	READ (u'r) list	WRITE (u'r) list
transfer on END= and ERR= can also be used with unformatted and direct access I/O statements	READ (u,f,END=n) list READ (u,f,ERR=n) list READ (u,f,END=n,ERR=n) list	WRITE (u,f,END=n) list WRITE (u,f,ERR=n) list WRITE (u,f,END=n,ERR=n) list

Where:

u	is a logical unit number, either an integer constant or variable
f	is a format reference (line number of a FORMAT statement or an array name)
list	is a list of I/O variables
r	is an integer variable or constant designating the record number
n	is a statement number
'	is the character used to separate the symbolic file number and the record number
,	is the character used to separate all other arguments of I/O statements

8.1.1 I/O Devices

PDP-11 FORTRAN supports all I/O devices supported by the Monitor. The default FORTRAN logical device assignments are described in Table 7-8. Device assignments can be altered by means of the ASSIGN routine. (See Section 7-14.2.3.)

Table 7-8

FORTRAN Logical Device Assignments

Logical Unit Number	Device
1	system disk, SY:
2	system disk, SY:
3	system disk, SY:
4	high-speed paper tape reader, PR:
5	line printer, LP:
6	terminal, KB:
7	system disk, SY:
8	BATCH system input device, BI:

8.1.2 I/O Records

All data is transmitted by input/output statements in terms of records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a teletypewriter a record is one line; for ASCII records, the amount of information is specified by the FORMAT reference and the I/O list; for unformatted binary records, the amount of information is specified by the I/O list; DEFINE FILE is used to specify record length for direct-access files.

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

```
READ 2,FIRST,SECOND,THIRD
```

is not necessarily equivalent to the following statements.

```
READ 2,FIRST
READ 2,SECOND
READ 2,THIRD
```

In this case at least three separate records are required, whereas the single statement

```
READ 2,FIRST,SECOND,THIRD
```

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

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record. Repositioning is not possible on all devices (see Section 7-8.6). If an input list requires more than one ASCII record of information, successive records are read.

8.1.3 I/O Lists

An input/output list contains the names of variables and array elements whose values are assigned on input or written on output. Input and output lists have the same form, consisting of individual elements separated by commas.

The simplest form of I/O list consists of a list of variable names or array element names. For example,

```
A,B,C
```

indicates three simple variable names.

An array name used alone in an I/O list specifies all of the array elements defined when the array is declared. Elements are read into core or output in the order specified for array storage (see Section 7-5.3.1). Thus, where the maximum element of the array X is X(4ØØ), an I/O list to include the entire array X could be written as

```
A,B,C,X
```

The elements of a list are specified in the order of their occurrence from left to right and may be enclosed in parentheses. The following list is identical in meaning to the previous list.

```
(A,B,C),X
```

An implied DO loop within the I/O list can be used to indicate transmission of only part of an array. This is performed by including a parenthesized quantity as a list element as follows:

```
          One-Dimensional Array        Two-Dimensional Array

          (A(I),I=n,m,k)               ((A(I,J),I=n,m,k),J=n',m',k')
or
          (A(I),I=n,m)                 ((A(I,J),I=n',m),J=n',m')
```

where A is an array name, I and J are array subscripts, and n, m, and k represent the initial, terminal and step values of the implied loop. When the step parameter is omitted it is presumed to be +1. For example,

```
        (X(K),K=1,4)
```

is equivalent to

```
        X(1),X(2),X(3),X(4)
```

The implied DO loop

```
        ((Y(I,J),I=1,2),J=1,3)
```

is equivalent to

```
        Y(1,1),Y(2,1),Y(1,2),Y(2,2),Y(1,3),Y(2,3)
```

As shown above, nesting is used to provide for various levels of indexing, up to three dimensions.

8.2 FORMATTED I/O

8.2.1 Formatted READ Statement

A formatted READ statement is of the form

```
            READ (u,f) list
            READ (u,f)
```

or

```
            READ f, list [equivalent to READ (4,f)]
```

where u indicates a logical unit number and f indicates a format reference (either a FORMAT statement or an array name whose elements contain a format specification).

A formatted READ statement causes information to be read from the specified device and placed in memory. The data is converted from external to internal format as indicated by the format specification. If an I/O list is provided, data items are stored as the values of the listed variables. If a list is not provided the data is stored as an alphanumeric string in the format itself. For example,

```
      READ (1,1ØØ) A,B,C
```

reads three values from device 1 according to the format in the FORMAT statement at line 1ØØ and assigns the values to the three variables named.

```
           READ (1,1ØØ)
     1ØØ   FORMAT (5HHELLO)
```

When the above READ statement is executed, and the characters ABCDE (for example) are given as input, the FORMAT statement at line 1ØØ becomes

```
     1ØØ   FORMAT (5HABCDE)
```

A READ statement of the form

```
           READ 11Ø,X,Y,Z
```

is similar to the first form discussed, except that input is assumed to come from logical unit 4 (default to PR:). The FORMAT statement associated with the operation is at line 11Ø.

Example:

```
           DIMENSION B(32)
     5Ø    READ (6,2ØØ) B
           .
           .
           .
    1ØØ    READ (6,B) I,J,K,L
           .
           .
           .
    2ØØ    FORMAT (32A4)
```

When statement 50 is executed in this example, the value of the array B is read from device 6. If the first two elements of B contain the following

```
      B(1) = '(4I6'
      B(2) = ')'
```

and the remaining elements of B, B(3) through B(32), are blank, the array B contains the format specification (4I6). (The format statement in line 2ØØ provides formatting information for the user to input enough data to completely fill the array B.) When statement 1ØØ is executed, the contents of the array B become the format reference for that READ statement. The variables I, J, K and L are read from device 6 according to the format 4I6 (that is, four 6-digit integers). This type of

READ statement (statement 1ØØ) is said to contain variable read-in formats and allows the user to specify an I/O format at run time (see Section 7-7.16).

Any format conversions and I/O control routines not needed in the resident section but required by overlay sections must be forcibly loaded into the resident section. This can be done by declaring the appropriate globals in an assembly language routine or inserting dummy FORMAT and input/output statements in the resident main program for all those routines needed in the overlays and not required in the resident section. See Section 7-9.10 for further details.

8.2.2 Formatted WRITE Statement

A formatted WRITE statement is of the form

```
       WRITE (u,f) list
```

or

```
       WRITE (u,f)
```

where u indicates a logical unit number and f indicates a format reference.

A formatted WRITE statement causes one or more logical records to be constructed and written on the designated device in ASCII format. A logical record contains up to 133 characters, including the carriage control character. The data is converted to external form as specified in the designated format reference. If an I/O list is provided, the values of the variables indicated are output. If a list is not provided, the information is read directly from the format reference and written on the device designated in ASCII form (this is generally done with Hollerith output). For example:

```
            WRITE (5,115) IERR,INET,ILOW
     115    FORMAT (3I4)
```

The above combination outputs three integer values to device 5.

```
            WRITE (5,12Ø)
     120    FORMAT (25H THIS IS HOLLERITH OUTPUT)
```

The above combination outputs 25 characters to form the message

```
            THIS IS HOLLERITH OUTPUT
```

on device 5 (the first character, blank, is not printed, but is used as a carriage control character).

8.2.3 PRINT Statement

The PRINT statement is of the form

```
PRINT f, list
```

where f indicates a format reference. The PRINT statement is a formatted WRITE statement whose output is always sent to the same device, the line printer. PRINT performs a WRITE (5,f). The default device can be changed by using the ASSIGN subroutine (see Section 7-14.2.4) or the ASSIGN Monitor command with respect to logical unit 5.

As an example,

```
PRINT 115, NA,NB,NC
```

outputs three values according to the format described in line 115. This statement is analogous to the third type of READ statement described in Section 7-8.2.1.

8.3 UNFORMATTED I/O

8.3.1 Unformatted READ Statement

An unformatted READ statement is of the form

```
READ (u) list
```

or

```
READ (u)
```

where u indicates a logical unit number.

An unformatted READ statement causes binary information to be read from the unit designated and stored in memory as the values of variables in the I/O list, if any. For example:

```
READ (7) A,B,C
```

This statement reads enough binary values from device 7 to fill three real variables (2 words each). If A, B, and C have been previously defined as double-precision variables, then three real values (4 words each) are read, and so on.

```
       READ (7)
```

allows the user to skip over one record on device 7. One such statement must appear for each record to be skipped.

Each READ statement reads one unformatted binary record. If the I/O list does not use all values in the record, the remaining values are discarded. If more values are required than are contained in the record, a run-time error message results.

Unformatted binary I/O generally is used to enable the storage of intermediate results from one program to another without the loss of precision involved in formatting data output. Unformatted I/O is performed between memory and such peripherals as paper tape, DECtape, magnetic tape or disk, but not to an ASCII device where such I/O is generally meaningless.

8.3.2 Unformatted WRITE Statement

An unformatted WRITE statement is of the form

```
       WRITE (u) list
```

or

```
       WRITE (u)
```

where u indicates a logical unit number.

An unformatted WRITE statement causes binary information to be read from memory and written on the device indicated in binary form. For example:

```
       WRITE (7) A,B,C
```

This statement writes the binary values of the variables A, B and C onto device 7.

```
       WRITE (7)
```

allows the user to write one null record on device 7. Execution of such a statement must appear for each null record to be output.

8.4 DIRECT ACCESS I/O

Direct access I/O can be performed on any directory-structured device.

8.4.1 DEFINE FILE Statement

The DEFINE FILE statement defines the record structure of a disk or DECtape file. However, DEFINE FILE automatically allocates a file if none exists. The DEFINE FILE statement can be used as either a declarative (nonexecutable) or executable statement in the program. Where DEFINE FILE is executable, arguments can be computed within the program.

The DEFINE FILE statement is formatted as follows:

```
DEFINE FILE u(m,n,U,ivar)
```

where the arguments are as follows:

u   is an integer constant or variable used as the logical unit number for this file specification.

m   is an integer constant or variable defining the number of records in the disk file. The records in the file are numbered from 1 to m.

n   is an integer constant or variable defining the length (in words) of each file record.

U   is a fixed argument designating that the disk file is unformatted (i.e., binary). No other characters are legal.

ivar is an integer variable, called the associated variable, which is set to point to the next record at the conclusion of an I/O operation on the file. If the associated variable is to be used by more than one subprogram or in an overlay environment, it should be placed in a resident common block.

The statement

```
DEFINE FILE 1(1ØØØ,1ØØ,U,IVAR)
```

specifies a 1ØØØ-record file on unit number 1, each record of which is 1ØØ words long.

The variable IVAR maintains an index of records processed, providing a pointer to the next record.

With respect to overlays, the DEFINE FILE routine uses storage space that must be allotted in such a way that the overlay system is not corrupted. In the statement

```
DEFINE FILE 1(1Ø,1Ø,U,INDX)
```

the variable INDX (the associated variable) is used to point to the record to be processed for statements such as the following ones.

```
READ (1'INDX)
WRITE (1'INDX)
FIND (1'INDX)
```

Every READ, WRITE or FIND done on unit 1 modifies INDX. INDX must be placed in the local storage of the resident section or in a resident common block. If it is not, the FORTRAN I/O system may destroy an unpredictable location in core when attempting to store into INDX.

8.4.2 FIND Statement

The format of the FIND statement is

```
FIND (u'r)
```

where u is the file specification as assigned by the DEFINE FILE statement, and r is an integer variable or constant specifying the record number. The FIND statement is included for compatibility with other FORTRAN systems. The Monitor does not support disk head positioning; therefore, the only effect of executing this statement is to update the associated variable for the specified unit.

8.4.3 Direct Access READ Statement

A direct access READ statement is of the form

```
READ (u'r) list
```

where u is a logical unit number as established in a prior DEFINE FILE statement and r is an integer variable or constant specifying the record number. An apostrophe is used to separate the two elements to differentiate this form of READ statement from a formatted READ statement. The list specifies the number of the items to be read.

A direct access READ statement provides random access to fixed-length records in a disk or DECtape file. For example,

```
READ (1'5Ø) ADC, ADX, AOD
```

reads record number 5Ø of the file on unit number 1 to obtain values for the variables ADC, ADX and AOD. Three 2-word values are transmitted since the variables indicated are real variables. If the variables were byte, integer, complex, or double precision, a corresponding number of bytes or words of data would be read for each variable.

As in other I/O operations, only the number of items in the list are transmitted regardless of the number of items in the record. If a list requires more values than are provided in the record, an error message is printed.

Implied DO loops can be included in direct access I/O statements.

8.4.4 Direct Access WRITE Statement

A direct access WRITE statement is of the form

```
WRITE (u'r) list
```

where u is a logical unit number and r is an integer variable or constant specifying the record number. An apostrophe is used to separate the two elements to differentiate this form of WRITE statement from a formatted WRITE statement. The list specifies the number of items to be output.

The direct access WRITE statement outputs a fixed-length record directed into a file and is analogous to the direct access READ statement described in the previous section.

8.5 TRANSFER OF CONTROL ON ERROR CONDITION

Both READ and WRITE statements can be written so as to transfer control to a given statement number if an error condition is detected. These variations can be specified in one of the following formats:

```
READ(u,f,END=n) list
READ(u,f,ERR=n) list
READ(u,f,END=,ERR=n) list
```

```
WRITE(u,f,END=n) list
WRITE(u,f,ERR=n) list
WRITE(u,f,END=n,ERR=n) list
```

where u is a logical unit designation, f is a format reference, and n is a line number within the program. The list is optional. END= and ERR= can also be used with unformatted and direct-access I/O.

The arguments END=n and ERR=n can appear separately, or together in the order illustrated. If an end-of-file condition is encountered during a READ, control transfers to the statement having the line number associated with the END parameter. If an END parameter is not specified, I/O on that device terminates and the program terminates with an error indication.

If an error is encountered on input or output, control transfers to the statement having the line number associated with the ERR parameter. If an ERR parameter is not specified, the program terminates with an error message.

If an error variable has been associated with a logical unit as a result of a CALL ASSIGN or CALL SETFIL statement, then when the ERR transfer is taken, the error variable will contain a code designating the type of the error. This code is the number of the error message within the Class 1 errors as described in Appendix K.

8.6 DEVICE CONTROL STATEMENTS

Refer to Table 7-13 which defines the type of I/O and the specific operations that can be performed on each of the peripherals.

8.6.1 BACKSPACE Statement

The BACKSPACE statement is of the form

```
BACKSPACE u
```

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device. The unit is repositioned to the beginning of the file opened on that unit, and the read/write mechanism spaces forward n-1 records, where n is the number of the record processed before the BACKSPACE. The effect is to backspace one ASCII record or one logical binary record, depending upon the device.

8.6.2 REWIND Statement

The REWIND statement is of the form

```
REWIND u
```

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device. This statement repositions the designated unit to the beginning of the currently opened file.

8.6.3 END FILE Statement

The END FILE statement is of the form

```
END FILE u
```

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device on which input or output operations are being performed. The END FILE statement causes the currently open file to be closed. END FILE also closes any open random-access files to allow another DEFINE FILE to be performed.

8.7 ENCODE AND DECODE

ENCODE and DECODE statements transfer data, according to format specifications, from one section of memory to another. DECODE changes data in ASCII format to the specified format. ENCODE changes data of the specified format into ASCII format. The two statements are of the form

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

where the arguments are as follows:

c the number of characters (bytes) to be converted.

f the FORMAT statement number or array specifying the format.

v the name of the array that contains, or is to contain, the ASCII character string; it must be an array name.

list the list of variables to be changed from or into ASCII format; it is the same as any other I/O list (see Section 7-8.1.3).

Example of the use of the DECODE statement:

```
          DIMENSION A(3),I(3)
          DATA A/'1234','5678','9Ø12'/
          DECODE (12,1ØØ,A)I
    1ØØ   FORMAT (3I4)
```

This code causes the integer array I to take on the numeric values of the three strings stored in the ASCII array, as follows.

```
I(1) = 1234
I(2) = 5678
I(3) = 9Ø12
```

DECODE is analogous to a READ statement, since it causes conversion from ASCII to internal format. ENCODE is analogous to a WRITE statement, causing conversion from internal data formats to ASCII.

The DECODE statement is used when it is necessary to process records having different formats. Figure 7-5 contains a segment of FORTRAN code that reads 80-character records from logical unit 1. The first two characters of each record contain an integer between 1 and 3 indicating the format of the remainder of the record. DECODE is used twice in the example: first to scan the record type field and then to read and translate the remainder of the record.

Note that the character data processed by ENCODE/DECODE must be in adjacent bytes in core. This means that if character data is to be stored in an integer array, the /ON (1-word integers) compile time switch must be used. If 1-word integers are undesirable then the character data should be stored in a real array.

```
      BYTE BUFF(8Ø),TITLE(12)
C
C     GET NEXT RECORD
5     READ(1,1Ø,END=9ØØ) BUFF
1Ø    FORMAT(8ØA1)
C
C     DECODE TYPE FIELD AND BRANCH ON IT
C
      DECODE(8Ø,2Ø,BUFF) ITYP
2Ø    FORMAT(I2)
      IF (ITYP.GE.1.AND.ITYP.LE.3) GOTO (1ØØ,2ØØ,3ØØ) ITYP
C
C     BAD RECORD
      WRITE(5,3Ø)BUFF
3Ø    FORMAT (' **BAD RECORD**'/1X,8ØA1)
      GO TO 5
C
C     ITYP=1
1ØØ   DECODE(8Ø,11Ø,BUFF)I,J,K
11Ø   FORMAT(T4,2I4,T24,I6)
        .
        .
        .
C
C     ITYP=2
2ØØ   DECODE(8Ø,21Ø,BUFF)A,B,C,D,I,J
21Ø   FORMAT(T2Ø,4F1Ø.5,2I5)
        .
        .
        .
C
C     ITYP=3
3ØØ   DECODE(8Ø,31Ø,BUFF)TITLE,I,J
31Ø   FORMAT(T3,12A1,T6Ø,2I5)
        .
        .
        .
C     END-OF-FILE TRANSFERS TO 9ØØ
9ØØ   CONTINUE
        .
        .
```

Figure 7-5

DECODE Example

PART 7

CHAPTER 9

OPERATING PROCEDURES

9.1 USING THE FORTRAN SYSTEM

Figure 7-6 outlines the steps required to prepare a FORTRAN source program for execution under the DOS/BATCH Monitor: (1) compilation, (2) linking, and (3) execution.

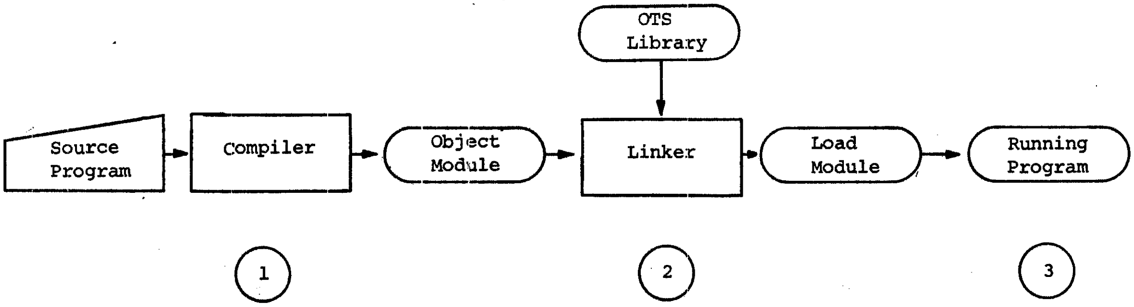


Figure 7-6

Steps in Compiling and Executing a FORTRAN Program

Step 1 in Figure 7-6 is initiated by a call to the FORTRAN Compiler, accompanied by a command string that describes input and output files, and switch options, if desired, to be used by the compiler. Step 2 is initiated by a call to the linker, accompanied by a similar command string. Steps 2 and 3 can be made to follow step 1 automatically by including the /GO switch option in the command string presented to the compiler. The /GO switch causes the linker to be called, the program linked, and execution begun in automatic sequence following compilation.

9.1.1 Filename Specifications

The DOS/BATCH FORTRAN compiler accepts a standard DOS command string of the following form.

```
output files < input files
```

The DOS/BATCH command string syntax is described more completely in Part 3 the DOS/BATCH Monitor.

The FORTRAN Compiler can produce two output files: an object module file and a listing file. The single input file permitted is a FORTRAN source file, containing one or more FORTRAN program units (a FORTRAN subprogram or main program). Each file specification is of the form

```
dev:filename.ext[uic]
```

where dev: is any legal device specification code. The usual DOS device specifications are shown in Table 7-9.

Table 7-9

Device Specification Codes

dev:	Device
SY:	system disk, assumed default device
DKn:	RK11 disk, cartridge unit n; n=∅ is the default condition
DC:	RC11 disk
DF:	RF11 disk
DPn:	RP03 disk, cartridge unit n; n=∅ is the default condition
KB:	user terminal
CR:	card reader
LP:	line printer
PR:	high-speed paper tape reader
PP:	high-speed paper tape punch
DTn:	DECtape unit n; n=∅ is the default condition
MTn:	Magtape unit n; n=∅ is the default condition
BI:	BATCH input data set

The filename may be any 1- to 6-character alphanumeric name. The filename extension may be any 1- to 3-character alphanumeric sequence. The filename extensions assumed or supplied on default for the FORTRAN Compiler (and for LINK) are shown in Table 7-10.

Table 7-10

Filename Extensions

File	Assumed Extension on Input File	Default Extension on Output File
FORTRAN Compiler		
object file	-	.OBJ
listing file	-	.LST
source file	.FTN	-
LINK		
load module	-	.LDA
object module	.OBJ	-

When a source file of the given filename (having no extension) cannot be found on the device specified using the default extension, the system attempts to find the filename with a null extension.

9.1.2 Compilation and Linking Procedures

To activate the FORTRAN Compiler under the DOS Monitor, the command

```
$RUN FORTRN
```

is given. The FORTRAN Compiler then prints FORTRAN and the compiler version number, and a # to indicate that it is ready to accept a command string.

A sample FORTRAN command sequence is shown below.

```
$RUN FORTRN
FORTRAN Vxxx
#OBJECT,LIST<FILE1
```

This command string directs the compiler to take the source file FILE1.FTN from the system device and output the files LIST.LST and OBJECT OBJ to the system device. The user then calls the DOS/BATCH Linker to link the object module(s) with the FORTRAN Library as follows.

```
     (1) Link                       (2) Link and Execute
     $RUN LINK                        $RU LINK
     LINK Vxxx                        LINK Vxxx
     #LOAD<OBJECT,FTNLIB/L/E          #LOAD<OBJECT,FTNLIB/L/GO
```

Input to LINK is one or more compiled object modules and any required library files. Output from LINK is a single load module and, optionally, a load map and symbol table file.

In the linkage shown in example (1) above, the linker output is a load module, which can then be loaded and run by means of the RUN command. The /L switch indicates that FTNLIB is a library file and the /E switch indicates the end of input to the linker. In example (2), the /GO switch causes the linker to output the load module to the system device, to load a copy of that load module, and to begin execution of the program, as though a

```
     $RUN LOAD
```

command had been given. When the /GO switch is used, the /E switch is omitted from the linker command string.

If any of the FORTRAN source files contains more than a single subroutine or a single main program, the resulting object module produced by the FORTRAN Compiler is a concatenated object module file. When such a file is linked by LINK, the linker /CC switch must be used following the object module file specification. Failure to use the /CC switch with a concatenated object module results in an error message from LINK and a defective load module.

The following sequence might be used to compile, link, and execute a FORTRAN system consisting of

a. the FORTRAN main program MAIN.FTN,
b. the FORTRAN subroutine SUBR1.FTN, and
c. several FORTRAN subroutines in the file UTILTY.FTN.

```
$RUN FORTRN
FORTRAN VØ6.Ø9
#MAIN,LP:<MAIN
#SUBR1,LP:<SUBR1
#UTILTY,LP:<UTILTY
#↑C
.KI
$RUN LINK
LINK Vxxx
#P1,LP:<MAIN,SUBR1,UTILTY/CC,FTNLIB/L/E
#↑C
.KI
$RUN P1
```

Either of the compiler output files can be eliminated by omitting its file specification from the command string. For example,

```
$RUN FORTRN
FORTRAN VØ6.Ø9
#FILE1<FILE1
```

produces FILE1.OBJ on the system device but no listing file, while

```
#,LP:<FILE1
```

produces a listing on the line printer, but no object module output.

9.1.3 Compile-Load-and-Go Operation

Rather than give separate command strings to both the FORTRAN Compiler and LINK, a single command string to the FORTRAN Compiler can be used to cause compilation, linking and execution. This one-step compile-load-and-go sequence is performed as follows:

```
$RUN FORTRN
FORTRAN Vxxx
#FILE1.LST<FILE1/GO
```

in which case the source program FILE1.FTN on the system device is compiled and linked to the FORTRAN Library (FTNLIB). A listing file (FILE1.LST), an object file (FILE1.OBJ) and a load module file (FILE1.LDA) are created on the system device. The load module file is then loaded and execution begun.

9.1.4 FORTRAN Library Usage

By means of the DOS Librarian, the user can construct his own libraries of machine language and FORTRAN routines, which he then searches at link time. The user is constrained to search all of his own libraries before searching the FORTRAN System Library.

Users should not add subroutines or functions to the DEC-supplied FORTRAN Library. Many routines in the OTS Library are order-dependent and the insertion of new routines could result in undefined global references when linking FORTRAN programs. Similarly, the deletion or rearrangement of routines in FTNLIB is likely to cause similar problems unless the user is familiar with the ordering dependencies of the FORTRAN Library. Therefore, users should create separate library files, using the Librarian. Thus, if MATLIB is a user library containing matrix manipulation routines, and the user writes a program (PROG) that uses routines from MATLIB, a command string to the Linker might be:

```
#PROG,LP:<PROG,MATLIB/L,FTNLIB/L/E
```

9.2 FORTRAN OUTPUT LISTING FORMAT

When a listing file is requested, each page contains a header with the following information:

Compile version identification
current time
current date
listing page number

Using the /LI:n switch option, the user can vary the type of listing file obtained; from a mere listing of error diagnostics to a complete listing of source, assembly language and symbol table. All listing files include block summaries describing the current program unit.

The block summary or descriptor block is a collection of information about a FORTRAN main program or subroutine, printed following the END statement on the source listing (before the assembly listing, if one is requested). Figures 7-7 and 7-8 show sample FORTRAN programs and their descriptor blocks. A descriptor block provides the following data:

a. A listing of the routines called from the program.

b. A listing of the switches specified in the command string.

c. A listing of the size of the program in both decimal words and octal bytes. Sizes are also provided for each common block declared within the program. Since common blocks may be shared among several programs, their sizes are of individual interest.

   The information is formatted into three columns under two headings: BLOCK and LENGTH. The name of the program or common block is listed under BLOCK, followed by the size in decimal words, followed by the size in octal bytes (in parentheses). These last two entries are under the LENGTH heading.

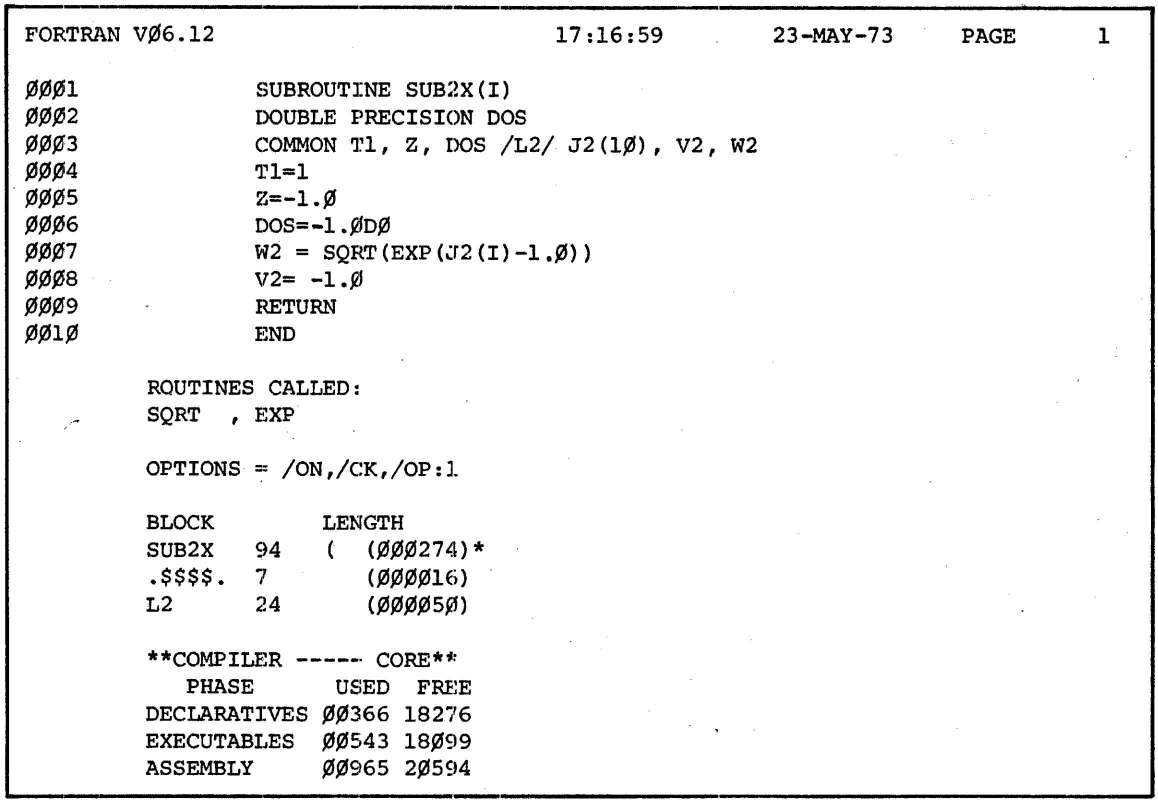
   The line containing the program name is followed by an asterisk. Where no program name is supplied, the name MAIN. is shown. The name .$$$$. denotes blank common; the name DATA. indicates a block data subprogram.

d. Following the Assembly listing, where one is requested, a final data item is supplied, entitled:

   COMPILER ----- CORE

   It has three subheadings: PHASE, USED, and FREE.

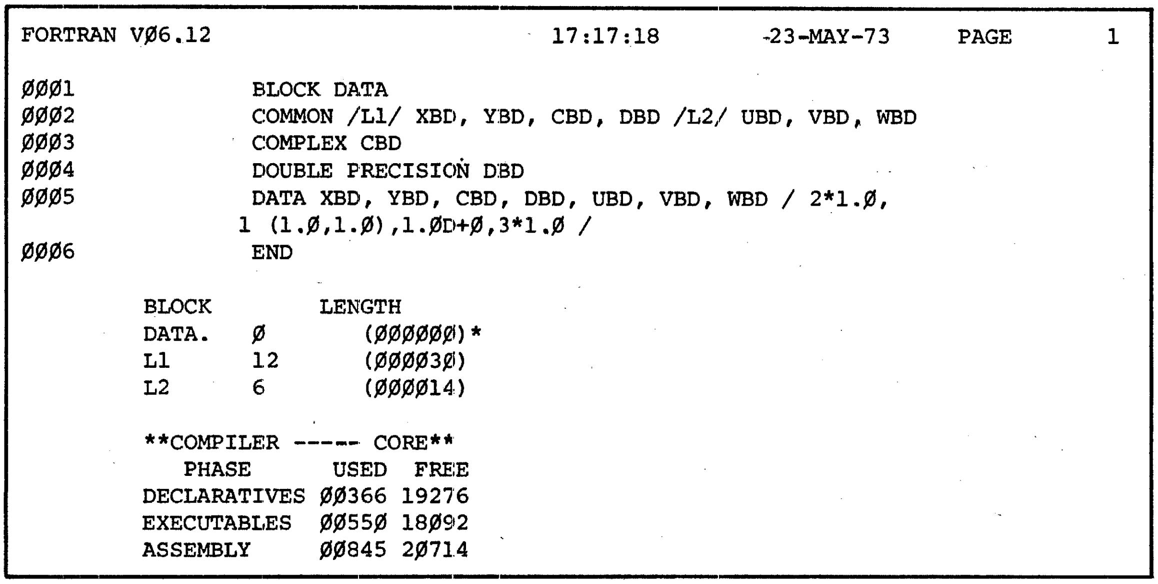
   These terms head columns that contain, respectively, the phase of the compiler (three phases). The number of decimal words of core storage used in that phase, and the number of decimal words of core storage free in that phase.



```
FORTRAN VØ6.12                          17:16:59        23-MAY-73     PAGE       1

ØØØ1                SUBROUTINE SUB2X(I)
ØØØ2                DOUBLE PRECISION DOS
ØØØ3                COMMON T1, Z, DOS /L2/ J2(1Ø), V2, W2
ØØØ4                T1=1
ØØØ5                Z=-1.Ø
ØØØ6                DOS=-1.ØDØ
ØØØ7                W2 = SQRT(EXP(J2(I)-1.Ø))
ØØØ8                V2= -1.Ø
ØØØ9                RETURN
ØØ1Ø                END

         ROUTINES CALLED:
         SQRT   , EXP

         OPTIONS = /ON,/CK,/OP:1

         BLOCK         LENGTH
         SUB2X    94   (  (ØØØ274)*
         .$$$$.   7       (ØØØØ16)
         L2       24      (ØØØØ5Ø)

         **COMPILER ----- CORE**
            PHASE       USED  FREE
         DECLARATIVES ØØ366 18276
         EXECUTABLES  ØØ543 18Ø99
         ASSEMBLY     ØØ965 2Ø594
```

Figure 7-7

Block Summary Example #1



```
FORTRAN VØ6.12                          17:17:18        -23-MAY-73    PAGE       1

ØØØ1                BLOCK DATA
ØØØ2                COMMON /L1/ XBD, YBD, CBD, DBD /L2/ UBD, VBD, WBD
ØØØ3                COMPLEX CBD
ØØØ4                DOUBLE PRECISION DBD
ØØØ5                DATA XBD, YBD, CBD, DBD, UBD, VBD, WBD / 2*1.Ø,
                  1 (1.Ø,1.Ø),1.ØD+Ø,3*1.Ø /
ØØØ6                END

         BLOCK         LENGTH
         DATA.    Ø       (ØØØØØØ)*
         L1       12      (ØØØØ3Ø)
         L2       6       (ØØØØ14)

         **COMPILER ----- CORE**
            PHASE       USED  FREE
         DECLARATIVES ØØ366 19276
         EXECUTABLES  ØØ55Ø 18Ø92
         ASSEMBLY     ØØ845 2Ø714
```

Figure 7-8

Block Summary Example #2

These sizes describe the symbol table space used in each phase and provide the user with a ratio of used to unused space in the compiler.

If the FORTRAN program was compiled without subroutine calls or switches, the information described in (a) or (b), respectively, is not provided. The information in (c) and (d) is provided as part of the listing for every FORTRAN program or subroutine.

9.3 COMPILE-TIME MEMORY REQUIREMENTS

The DOS/BATCH FORTRAN Compiler uses all available memory space for the program symbol table. A simple variable entry in the symbol table is eight words long. A constant entry is eight words plus the size of the constant. An array entry is ten words plus one word for each dimension. For example, a complex constant entry is twelve words long and a 3-dimensional array entry is thirteen words long. The user can reduce the amount of memory used by the compiler by his choice of input/output files in command strings and by reducing the number of continuation lines permitted. Memory space saved in this way is available as additional symbol table space.

The /CO switch, which controls the number of allowable continuation lines on a given FORTRAN statement, takes space from the symbol table area. Thus, /CO:99 takes the maximum amount of space from the symbol table while /CO:Ø makes additional space available to the symbol table. (The default value is /CO:5).

The devices used for I/O affect the compiler core requirements. The minimum core required for I/O operations would be used with the command sequence

```
$RUN FORTRN
FORTRAN Vxxx
#FILE<FILE
```

where source and object files are on the system device. Memory requirements are increased, and symbol table space correspondingly reduced, when the user requests listings or uses devices other than the system device. For example,

```
#FILE,KB:<DT:FILE
```

requires considerably more memory space than the previous command string.

PART 7

CHAPTER 10

FORTRAN OPERATING ENVIRONMENT

1Ø.1 FORTRAN OBJECT TIME SYSTEM

The FORTRAN Object Time System (OTS) is composed of the following:

a. Mathematics routines, including all standard FORTRAN functions plus other arithmetic routines needed to do arithmetic operations (e.g., floating point).

b. Miscellaneous utility routines (PDUMP, SETERR and SETFIL).

c. I/O Routines, which handle the various types of FORTRAN I/O.

d. Error handling routines, which process arithmetic errors, I/O errors and system errors.

e. Miscellaneous (Polish) routines required by the compiled code ($SBS, $DOEND, $POP, etc.).

The library is designed as a large number of small pieces so that unnecessary routines can be omitted at link time. Thus, if the user performs only sequential formatted I/O, none of the random access routines are linked to his program.

1Ø.2 THREADED CODE

Most FORTRAN compiled statements generate calls to library subprograms. These calls are based on the technique of evaluating expressions in Polish notation, which breaks down expressions into a large number of simple operations performed in a linear sequence. These operations use the PDP stack for evaluating all expressions and subexpressions.

The implementation of Polish notation makes several assumptions.

a. The first operation done in a Polish sequence is always a "push".

b. It is not necessary to place breakpoints (as in ODT) in the middle of an arithmetic statement.

c. Speed does not suffer by assignment of a register (R4) for special purposes.

As an example, the FORTRAN statements

```
A=1.Ø
B=1.Ø
```

generate the following code.

```
;A=1.Ø
$PØØØ1
.GLOBL $POP3
$POP3,A
;B=1.Ø
$PØØØ1
$POP3,B
```

Routines such as those above, referenced directly or indirectly, are inserted into the user program and are found at the end of the assembly listing; other routines are linked to the user program from the FORTRAN Library. The routines inserted into the user program as a result of the above code follow.

```
$PØØØ1:  MOV       #$RØØØØ+4,RØ      ;GET ADDRESS OF VALUE
         BR        $FØØØ1

$RØØØØ:  Ø4Ø2ØØ                      ;FLOATING-POINT CONSTANT 1.Ø
         ØØØØØØ

$FØØØ1:  MOV       -(RØ),-(SP)       ;PUSH 2-WORD VALUE ONTO
         MOV       -(RØ),-(SP)       ;STACK
         JMP       @(R4)+            ;GO TO NEXT ROUTINE
```

The routine $POP3 is in the FORTRAN Library. $POP3 pops a real value off the stack into the memory location whose address follows the call to $POP3 in the threaded code.

```
$POP3,A
```

pops the value on top of the stack into the two memory words reserved for A. Similarly,

```
$POP3,B
```

places the 2-word value found on top of the stack into B. As another example, the statement

```
A =B+C
```

generates code similar to the following

```
$PØØØ1              ;PUSHES THE VARIABLE B ONTO THE STACK.
$PØØØ2              ;EACH OPERATION CONSISTS OF THE ADDRESS OF A
.GLOBL $ADR         ;ROUTINE TO BE EXECUTED.  A PUSH PLACES A
$ADR                ;VALUE ON THE STACK; A POP REMOVES A VALUE.
$POP3,A
.+2                 ;THIS LINE CAUSES AN EXIT FROM POLISH MODE;
                    ;NORMAL EXECUTION IS RESUMED.
```

$PØØØ1 and $PØØØ2 push the values of B and C onto the stack; $ADR is a FORTRAN Library routine to add two floating-point numbers; $POP3 saves the result in the variable A. The subroutine calls for the above sequence follow.

```
$PØØØ1: MOV     #B+4,RØ      ;GET THE ADDRESS OF B.
        BR      $FØØØ1       ;JUMP TO PUSH.
$PØØØ2: MOV     #C+4,RØ      ;GET THE ADDRESS OF C.
$FØØØ1: MOV     -(RØ),-(SP)  ;PUSH TWO
        MOV     -(RØ),-(SP)  ;WORDS ON STACK.
        JMP     @(R4)+       ;JUMP TO NEXT ROUTINE.
$POP3:  MOV     (R4)+,R3     ;GET ADDRESS OF
                             ;VARIABLE DESTINATION.
        MOV     (SP)+,(R3)+  ;POP A VALUE
        MOV     (SP)+,(R3)+  ;FOR THE VARIABLE.
        JMP     @(R4)+       ;GO TO NEXT ROUTINE.
```

Note that the instruction

```
JMP     @(R4)+
```

jumps to the next routine in the list as well as incrementing R4 over that item in the threaded code. All internal functions are called in this manner and exit using such a jump instruction and must clear any stack space used (except for the return value left on top of the stack).

All routines explicitly called by the user (i.e., subroutines and external functions) are called using the PDP-11 subroutine calling convention (see Chapter 16).

1Ø.2.1 Entry to Polish Mode

Entry to Polish mode is made via a call to the $POLSH routine, as follows.

```
JSR     R4,$POLSH
```

This invokes the following routine.

```
        $POLSH:  TST         (SP)+               ;DELETE OLD VALUE OF
                                                 ;R4 PUSHED ON ENTRY
                 JMP         @(R4)+              ;EXECUTE NEXT THREADED CODE CALL
```

The word following the JSR to $POLSH is the first word of Polish code to be executed.

1Ø.2.2 Exit from Polish Mode

An explicit exit from Polish mode is made by means of a word containing the address of the following word. For example:

```
          .
          .
          .                       ;IN POLISH MODE
        .WORD    .+2              ;LEAVE POLISH MODE
          .                       ;CONTROL PASSES TO THIS LOCATION
          .
          .
```

1Ø.2.3 Polish Mode Subroutine Calls

The PDP-11 FORTRAN calling sequence convention is described in Chapter 7-15. The PDP-11 FORTRAN Compiler (VØ6) implements this convention by means of Polish Mode operators described in this section.

A subroutine or function call is performed using the $CALL service routine followed by one argument (the routine to be called). The argument list follows immediately in memory. The routine called is entered in direct execution mode. (This is true of FORTRAN compiled subprograms as well as assembly language programs.)

Thus, it is necessary to re-establish Polish mode after each subroutine or function call using one of the library routines $RPOLO or $RPOLN. These routines are similar to $POLSH but $RPOLN additionally adjusts the processor stack register to delete any temporary arguments created for the call. For example the statement

```
        X = FNC(A+B,C)
```

will result in the following code.

```
        $PØØØ1                          ;PUSH VALUE OF A ON STACK
        $PØØØ2                          ;PUSH VALUE OF B ON STACK
        $ADR                            ;ADD REAL -- RESULT LEFT ON STACK
        $SVSP,$FØØØ1                    ;COPY ADDRESS OF A+B INTO ARG LIST
        $CALL,FNC
        BR      .+6                     ;TWO ARGUMENTS
$FØØØ1: Ø                               ;FILLED IN AT EXECUTION TIME
        +C                              ;ASSEMBLED IN ADDRESS OF C
                                        ;ARRIVE HERE IN NON-POLISH MODE
        JSR     %4,$RPOLN               ;RETURN TO POLISH MODE AND
        +4                              ;DELETE TEMPORARY VALUE A+B
```

1Ø.3 FORTRAN RUN-TIME MEMORY ORGANIZATION

The memory map during execution of a FORTRAN program is shown in Figure 7-9.

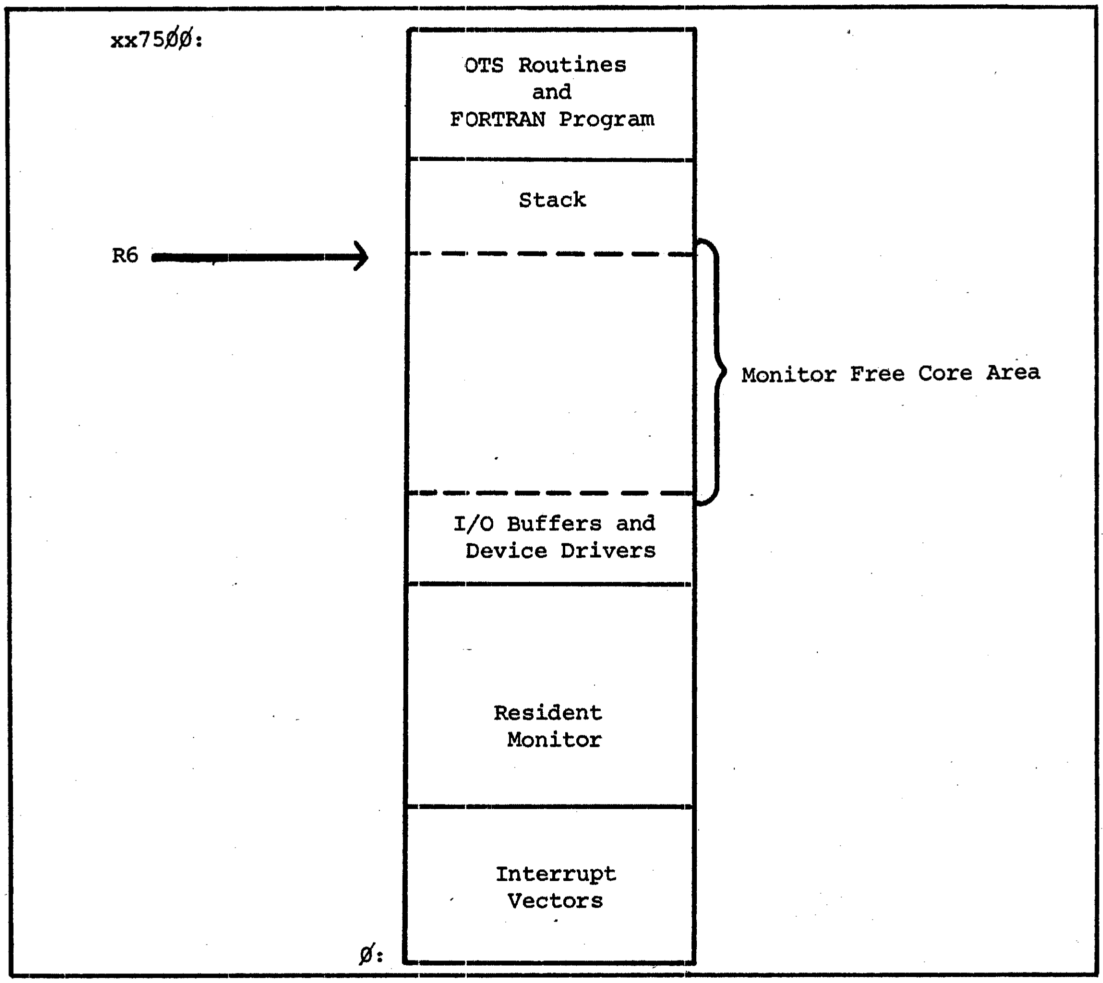


Figure 7-9

Memory Map Organization

As files are opened, space is taken from the Monitor free-core area to accommodate additional I/O buffers and device drivers. The FORTRAN program and OTS routines use the processor R6 stack for storage of temporary results. Overflow of the stack into the areas dynamically allocated by the Monitor may cause DOS fatal errors or system crashes.

I/O buffer space and device drivers are released and the memory reclaimed when the logical unit is closed by an END FILE statement. END FILE may be used on both input and output files.

DOS I/O uses space from the free core area as follows:

1. One device driver routine for each device that has been initialized (.INIT) but not released (.RLSE). (The END FILE statement performs a .CLOSE and .RLSE). The approximate DOS driver sizes are given in Table 7-11.
2. One I/O buffer for each open dataset (FORTRAN logical unit in use). The buffer size depends upon the device.
3. One Device Data Block (DDB) for each open dataset (16 words).
4. One File Information Block (FIB) for each open file (16 words).
5. One bit map for each unit on which a file is open for output (64 words).

Table 7-11

Approximate DOS Device Driver Sizes

Device Specifier	Device Driver Size (Words)	I/O Buffer Size (Words)
DK	130	256
DF	70	64
DC	70	64
DP	260	512
DT	190	256
MT	460	256
LP	160	48
CR (ASCII only)	210	48
KB	600	32
PP	80	32
PR	70	32

1Ø.4 DEBUGGING FORTRAN PROGRAMS

The use of traditional FORTRAN debugging techniques such as PDUMP is recommended for development of FORTRAN applications (see Chapter 7-14).

System error reporting and traceback information is of significant value in debugging. Figure 7-1Ø is an example of the traceback feature in the FORTRAN system.

At run time, diagnostics are printed by OTS with a trace of the flow of control within the user-written code. Following each error code are printed the error headings NAME and SEQ, below which are the names of the routines through which the call is being traced and the sequence number of the specific line in which the error occurred (or that from which that subprogram was called). The first name and sequence number at the top of the list is the error location; subsequent names and numbers refer to the path through which the program reached that point.

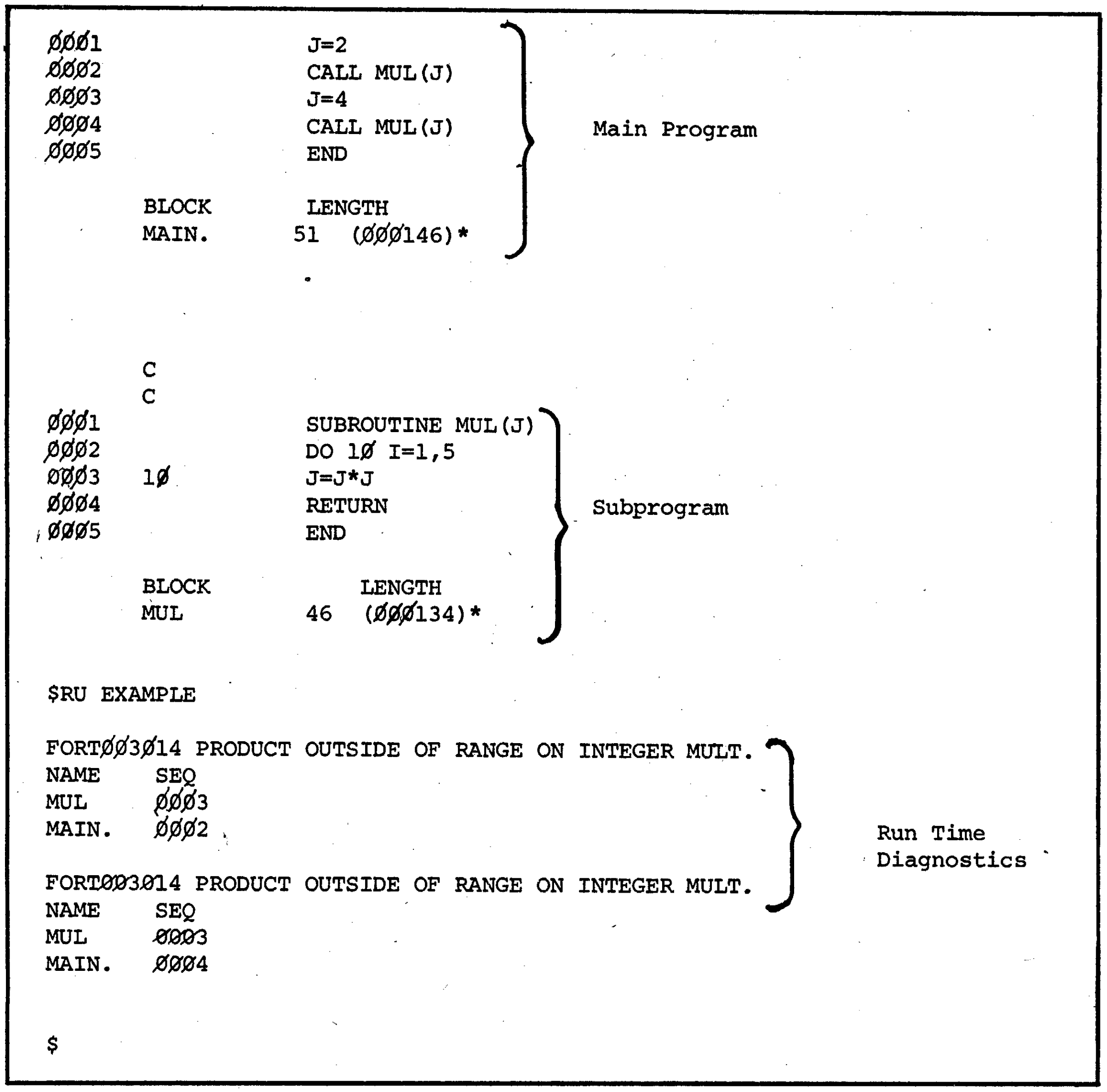
The example in Figure 7-20 shows the same error message being printed twice. Though the two errors were generated by the same line in the subprogram, they are traced to different lines in the main program.

It requires considerable experience to successfully use ODT with a FORTRAN program. ODT was not intended to handle the problems of FORTRAN debugging; it is especially hard to use when trying to debug threaded code.

The threaded code generated by the compiler is a sequence of addresses, rather than machine instructions. ODT breakpoints can be placed only on machine instructions. The user is therefore constrained to use breakpoints only in places where the code leaves Polish Mode; e.g., a subroutine or function call. Alternatively, breakpoints can be placed in the Polish routines themselves.

There is one significant inconvenience associated with putting breakpoints in Polish routines. Polish routines are usually called from several places in a program and when the breakpoint in a Polish routine is encountered, the user must look at R4 to find the address from which the routine was called.

When executing a FORTRAN program that stores data outside the memory area allocated for an array, the Monitor messages F342 and F344 may occur and the system may halt. The compiler /CK switch (Appendix K) causes all subscript references to be checked for upper and lower bounds. Use of /CK is recommended while debugging FORTRAN programs.



```
ØØØ1                J=2
ØØØ2                CALL MUL(J)
ØØØ3                J=4
ØØØ4                CALL MUL(J)
ØØØ5                END

       BLOCK        LENGTH
       MAIN.       51  (ØØØ146)*

       C
       C
ØØØ1                SUBROUTINE MUL(J)
ØØØ2                DO 1Ø I=1,5
ØØØ3   1Ø           J=J*J
ØØØ4                RETURN
ØØØ5                END

       BLOCK            LENGTH
       MUL          46  (ØØØ134)*

$RU EXAMPLE

FORTØØ3Ø14 PRODUCT OUTSIDE OF RANGE ON INTEGER MULT.
NAME     SEQ
MUL      ØØØ3
MAIN.    ØØØ2

FORTØØ3Ø14 PRODUCT OUTSIDE OF RANGE ON INTEGER MULT.
NAME     SEQ
MUL      ØØØ3
MAIN.    ØØØ4


$
```

Main Program
Subprogram
Run Time Diagnostics

Figure 7-1Ø

Example of Run Time Diagnostics

1Ø.5 FORTRAN OTS ERROR PROCESSING

The FORTRAN OTS error diagnostics are divided into nine classes (Classes Ø-8). The messages associated with each diagnostic are given in Appendix K.

The FORTRAN OTS maintains an error class table, which governs the actions taken for each error class. The user can modify the values in the error class table (and hence the error processing) by means of the SETERR subroutine call (Chapter 7-14).

Each error class entry contains values for the following variables:

COUNT — a count of the number of errors that have occurred for this class.

FLAG — set to 1 on any occurrence of an error in the class. May be tested and reset by means of CALL TSTERR (Chapter 7-14).

MAX — an integer specifying the action to be taken on an error in the class.

The actions taken by the OTS when an error condition occurs are determined by the value of MAX as follows:

MAX>Ø — Set FLAG, log error message on output device, increment COUNT, and call EXIT if COUNT>MAX.

MAX=Ø — Set FLAG, log error message on output device, and continue program execution.

MAX=-1 — Set FLAG, do not log message, ignore count, and continue program execution.

MAX=-2 — Do not log message, EXIT to Monitor, and print I351 message.

MAX=-3 — Immediate exit to Monitor with a message of the form:

```
FØ3Ø ØCCØNN
```

where CC and NN are the error class and number (in octal) of the condition causing the exit (see Appendix K).

The error classes and default MAX values are detailed in Table 7-12.

Table 7-12

FORTRAN OTS Error Classes

Class	Default MAX Value	Meaning
Ø	-3	Fatal errors; it is recommended that the occurrence count (-3) not be changed.
1	Ø	Physical I/O errors.
2	1	Errors in FORMAT statements.
3	3	Arithmetic overflow or division by 0.
4	4	Incorrect arguments to library functions or subroutines.
5	-1	Arithmetic underflow errors.
6	Ø	Conversion errors.
7	7	Subscripting errors.
8	1	Errors involving incorrect linkages to OTS routines; the maximum occurrence count (1) cannot be changed, although no diagnostic is given.

For example, arithmetic underflows (class 5) are ordinarily ignored by the FORTRAN OTS; arithmetic overflows (class 3) are logged and the default error limit is three occurrences.

PART 7

CHAPTER 11

FORTRAN MONITOR I/O CONSIDERATIONS

Input/output operations for a FORTRAN-compiled program are performed by the FORTRAN Object Time System. The FORTRAN language statements related to I/O are described in Chapters 7 and 8. It is the purpose of this Chapter to describe the processing of I/O requests in greater detail, with emphasis on the FORTRAN OTS and DOS/BATCH Monitor mechanisms and representations. File formats and interactions between DOS and the FORTRAN OTS are also described.

The FORTRAN OTS accomplishes all physical I/O through the DOS Monitor. FORTRAN I/O is therefore device-independent to the extent that such independence is supported by the DOS/BATCH Monitor. For example FORTRAN formatted WRITE statements can be directed to any ASCII output device, but direct-access I/O can only be done on devices having fixed-length addressable blocks (disks and DECtape, but not industry-standard magnetic tape).

11.1 INPUT/OUTPUT OVERVIEW

As described in Chapter 8, all FORTRAN I/O operations are done in terms of logical records. Each READ or WRITE statement transmits one or more logical records. The overall flow of control in processing a single FORTRAN READ or WRITE statement appears in Figure 7-11.

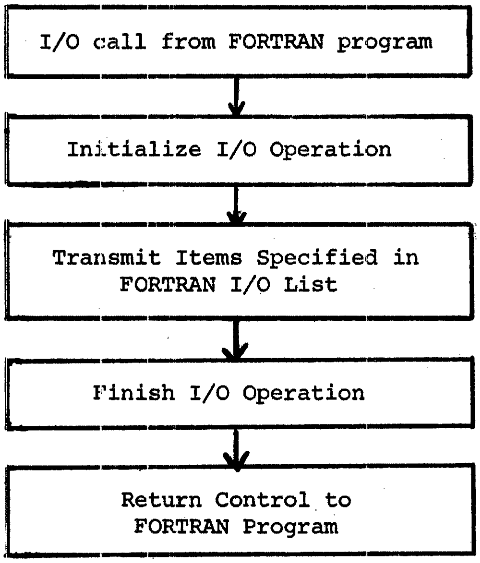

I/O call from FORTRAN program
Initialize I/O Operation
Transmit Items Specified in FORTRAN I/O List
Finish I/O Operation
Return Control to FORTRAN Program

Figure 7-11
FORTRAN I/O Flow

The operations accomplished follow.

1. Initialize I/O Operation:

   a. Verify legal unit number, obtain $DEVTB entry.

   b. Test to see if the file is open; if it is not, open the file.

   c. If the file is open, check for compatibility of device and file for the type of I/O requested.

   If a contiguous file is to be allocated, this is done as a part of the file-opening process in step (b) above.

2. Item Transmission:

   a. If the operation is a READ, obtain a record if needed; check for I/O errors.

   b. Transmit items according to the I/O list. If the record length is exceeded, issue a diagnostic message.

3. Finish I/O Operation:

   a. Complete the current record with padding and record separators as required.

   b. If the operation is a WRITE, transmit the record and wait for completion.

   c. Evaluate physical I/O error returns.

In the above descriptions, a file is not opened until a READ or WRITE statement for that unit is executed. Thus, errors such as nonexistent files or inability to locate a contiguous file are not detected until an I/O operation is attempted.

All FORTRAN I/O is single-buffered. A DOS .WAIT macro is issued after each Monitor I/O call. All transfers of control on END= and ERR= precisely identify the FORTRAN I/O statement causing the END= or ERR= transfer.

The principal data structure used by the FORTRAN OTS for controlling I/O is the device table, $DEVTB, described in Sections 7-11.4 and 7-11.5. The unit number supplied in a READ or WRITE statement is used to select a device table entry. The device table entry contains information for each logical unit, including:

Filename.
Physical device specification and unit number.
User identification code (UIC).
Protection code.
Open/close status.

FORTRAN logical unit defaults are specified by the values assembled into $DEVTB in the version supplied with the FORTRAM OTS. CALL SETFIL and CALL ASSIGN (Chapter 7-14) can be used to modify $DEVTB entries under program control. The DOS ASSIGN command can be used to override some of the $DEVTB attributes.

The physical devices supported by FORTRAN and DOS, and the types of FORTRAN operations permitted on each device, are specified in Table 7-13.

Table 7-13

DOS/BATCH FORTRAN Standard Peripheral Devices

Name	Device Specification	FORTRAN I/O Type			FORTRAN I/O Operations			
		Formatted	Unformatted	Direct-Access	READ	WRITE	ENDFILE	BACKSPACE REWIND
Disk*	DC: DF: DK: DP: } or Sy:	X	X	X	X	X	X	X
DECtape	DTn:	X	X	X	X	X	X	X
Magtape	MTn:	X	X		X	X	X	X
Line Printer	LP:	X				X	X	
Card Reader	CR:	X	X		X		X	
Terminal	KB:	X			X	X	X	
High-speed paper tape reader	PR:	X	X		X		X	
High-speed paper tape punch	PP:	X	X			X	X	
Low-speed paper tape reader/punch	PT:	X			X	X	X	

* The universal mnemonic SY can be used to specify the system disk regardless of whether that unit is an RC11, RF11, RK11, RP∅2, or RP∅3 disk.

X Indicates that a specific I/O mode or operation is supported on the indicated device.

11.2 FILE STRUCTURES AND FORMATS

FORTRAN input/output facilities are provided by three packages of OTS routines: formatted I/O, unformatted I/O, and direct-access I/O. General characteristics of DOS file structures and the three I/O packages are discussed in following sections.

The DOS Monitor provides a variety of I/O modes, and two principal file structure organizations:

a. Linked files for sequential access.

b. Contiguous files for direct or sequential access.

Linked files consist of a series of blocks, which need not be physically contiguous on the device. Each block contains a pointer to the next block of the file. When a linked file is initially opened for output, the first block is allocated. As output requests occur, additional blocks are allocated and linked to the file. The length of the file is therefore unknown at the time the file is opened.

Formatted and unformatted output ordinarily uses linked files. A contiguous file may be used for output following a file allocation operation through SETFIL.

Contiguous files are specifically intended for direct-access I/O. A fixed-size contiguous file must be initially allocated on disk or DECtape by means of DEFINE FILE or CALL SETFIL. Direct-access I/O statements may then read or write individual records without having to read all preceding records.

Table 7-14 summarizes the file structures and DOS I/O modes used by the FORTRAN I/O packages.

Table 7-14

File Structures and I/O Modes

FORTRAN I/O Type	DOS File Structure	DOS I/O Mode
Formatted	Linked or Contiguous	Formatted ASCII Normal
Unformatted	Linked or Contiguous	Formatted Binary Normal
Direct-access	Contiguous	Unformatted Binary

11.2.1 FORTRAN Formatted I/O

The formatted input/output routines read or write variable-length formatted ASCII records. A record consists of a maximum of 133 ASCII characters transmitted under control of a format specification, followed by record separator characters (typically carriage return/line feed). For example, the format specification 133A1 transmits a maximum length record.

On output to a nonprinting device (e.g., disk file), the record separator consists of the carriage return/line feed sequence. The maximum length output line thus requires 135 bytes on a disk file. The file consists simply of a sequence of output lines; there are no additional headers, checksums, or byte counts.

On output to a printing device (KB: or LP:) the record separator appended to each record consists of a carriage return and a vertical tab. The first character of each record is deleted from the record and is interpreted as a carriage control character; it may cause output of Ø, 1 or 2 line feed characters or a form feed character.

On input, records longer than 135 bytes (including separator) are truncated. For shorter records, the last character (line feed, form feed or vertical tab) is deleted. The next-to-last character is deleted if it is a carriage return. No other special interpretations are provided; that is, form feeds, tabs and other special ASCII characters are passed to the program as single characters. In particular, tab characters are not converted to blanks.

Note that interpretation of the carriage control character is provided by the FORTRAN OTS, not by the Monitor or utility programs. If output containing FORTRAN carriage control characters is directed to a disk or tape file for later printing, that printing must be done under control of a FORTRAN program in order to obtain the desired carriage control.

11.2.2 FORTRAN Unformatted I/O

The unformatted input/output routines read or write variable length binary records using the DOS/BATCH formatted binary mode. Each record contains checksums, for parity checking, and a byte count.

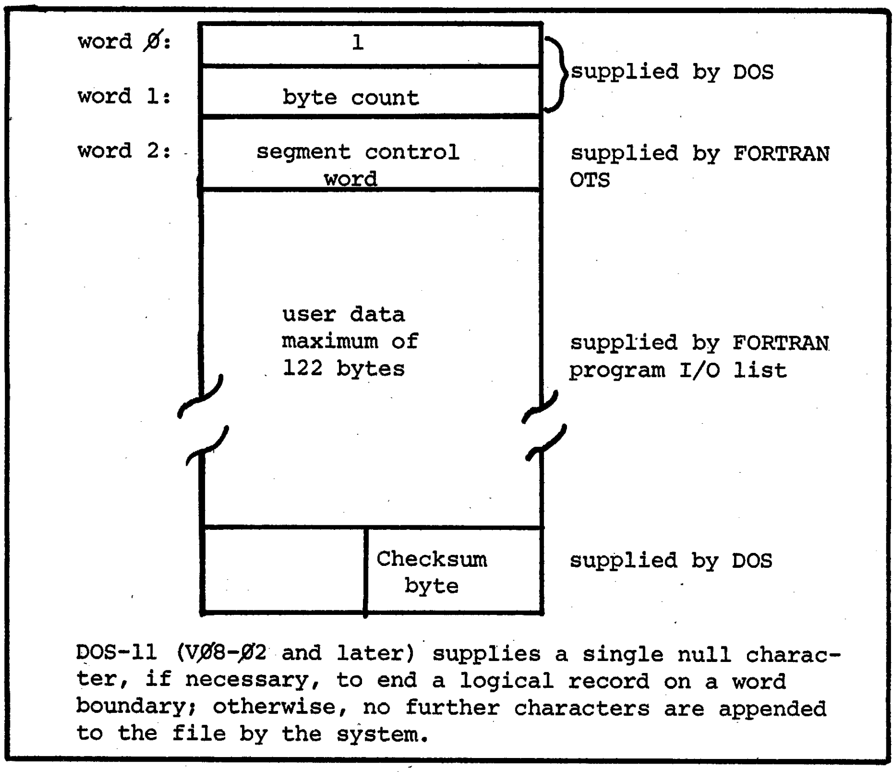
A FORTRAN unformatted WRITE statement outputs a single logical record. A logical record consists of one or more segments, each segment containing a maximum of 61 words of data from the user program and a segment control word supplied by the FORTRAN OTS. The segment control word is inserted as the first data word of the segment and may take one of the values shown in Table 7-15.

Table 7-15

Segment Control Word

Value	Meaning
Ø	Neither first nor last segment
1	First segment of logical record
2	Last segment of logical record
3	First and last segment of logical record (Single-segment record)

Each segment of the logical record is output as a DOS formatted binary line. The format of each segment is shown in Figure 7-12.



DOS-11 (VØ8-Ø2 and later) supplies a single null character, if necessary, to end a logical record on a word boundary; otherwise, no further characters are appended to the file by the system.

Figure 7-12

Logical Record Segment Format

Each segment has four words of overhead; the user must allow for these overhead words when allocating a contiguous file for unformatted I/O through CALL SETFIL.

11.2.3 FORTRAN Direct-Access I/O

The direct-access I/O routines read or write fixed-length binary records in a contiguous file. The DOS unformatted binary I/O mode is used. The maximum record length is 32,767 bytes.

The logical record structure for a direct-access file is determined entirely by the DEFINE FILE statement. The records themselves contain no checksums, byte counts, or record separators. Different programs can process the same file using different logical record structures. In Figure 7-13, Program 1 operates on records consisting of 1Ø x 1Ø integer matrices, while Program 2 operates on records consisting of a single row of a matrix.

```
Program 1
       INTEGER MATRIX(1Ø,1Ø)
        .
        .
        .
       DEFINE FILE 1 (2Ø, 1ØØ, U, INDX)
        .
        .
        .
C      WRITE OUT A 1Ø x 1Ø MATRIX
       WRITE (1'NREC) ((MATRIX(I,J), J=1,1Ø) ,I=1,1Ø)
        .
        .
        .
       END
Program 2
       INTEGER ROW(1Ø)
        .
        .
        .
       DEFINE FILE 1(2ØØ, 1Ø, U, INDEX)
        .
        .
        .
C      READ A 1Ø ELEMENT ROW OF MATRIX
       READ(1'NREC) ROW
        .
        .
        .
       END
```

Figure 7-13
Program Example Using Logical Records

If a WRITE statement transmits fewer words than will fit in the record, the remainder of the record is filled with zeros.

The direct-access record structure is independent of the physical block size of the I/O device, but more efficient operation results if the record size is an exact divisor or multiple of the physical block size (RF and RC disks = 64 words, RPØ2 or RK disk and DECtape = 256 words, and RPØ3 disk = 512 words).

11.3 FORTRAN I/O ERROR HANDLING

Three classes of run-time error conditions are specifically associated with FORTRAN I/O operations.

Class 1	Nonexistent file, physical I/O error, etc.
Class 2	FORMAT syntax errors
Class 6	Conversion errors during format conversions.

A complete list of FORTRAN run-time errors is contained in Appendix K.

As described in Chapter 8, READ and WRITE statements can specify transfer of control on an end-of-file condition (END=) and on certain physical I/O conditions (ERR=). The SETFIL and ASSIGN subroutines (Appendix F) can specify an error variable, IERR, which is set to indicate the specific type of I/O error.

Errors in classes 2 and 6 are processed as described in Section 1Ø.5 and Appendix K. Class 1 I/O error conditions are processed as follows:

Assume the error condition is identified by the diagnostic FORTØØ1ØNN.

1. Set the IERR variable, if specified in the previous SETFIL to NN.
2. If END=n is specified, and the error condition is an end-of-file (FORTØØ1ØØ4), transfer control to statement n.
3. If ERR=n is specified, transfer control to statement n.
4. Otherwise, process the error according to Section 7-1Ø.5.

11.4 FORTRAN DEVICE ASSIGNMENTS

The default FORTRAN device assignments are shown in Table 7-16. These assignments can be temporarily altered through use of a call within the user program to the SETFIL or ASSIGN routines or by use of the ASSIGN Monitor command. Section 7-11.5 describes how to permanently alter the FORTRAN device assignments.

Table 7-16

FORTRAN Logical Device Assignments

Logical Unit Number	Device	Default File Name
1	system disk, SY:	FORØØ1.DAT
2	system disk, SY:	FORØØ2.DAT
3	system disk, SY:	FORØØ3.DAT
4	high-speed paper tape reader, PR:	FORØØ4.DAT
5	line printer, LP:	FORØØ5.DAT
6	terminal keyboard, KB:	FORØØ6.DAT
7	system disk, SY:	FORØØ7.DAT
8	BATCH system input device, BI:	FORØØ8.DAT

Device assignments are determined in one of three ways. These are described below in order of increasing priority.

a. The device table default values described in Appendix D.

b. A call within a FORTRAN program to the SETFIL (or ASSIGN) subroutine (see Chapter 7-14) overrides the default device assignments.

c. The Monitor ASSIGN command overrides both default assignments and any assignments made within the program by a call to the SETFIL routine.

The SETFIL routine causes entries to be made in the FORTRAN device table, $DEVTB, which is described in greater detail in Section 7-11.5.

An ASSIGN Monitor command for use with FORTRAN is specified in the form:

```
$ASSIGN dev:file.ext[uic], n
```

where the $ character is printed by the system; dev:file.ext[uic] represents a file specification, and n represents a FORTRAN unit number. For example,

```
$ASSIGN DT:,2
```

forces a search on DECtape unit Ø for the filename (and extension) associated with logical unit 2 in the device table (or in a SETFIL call).

```
$ASSIGN DT3:FILE.TMP,3
```

causes the file FILE.TMP on DECtape unit 3 to be associated with logical unit 3.

Alterations in the device table made by a call to the SETFIL subroutine are only valid during the program in which they appear. Device assignments made with DOS ASSIGN command vary in scope depending upon when the command is given.

a. When no program is in core, assignments made through an ASSIGN command are valid through several runs of the same or different programs; they can be removed by an ASSIGN command with no arguments. For example:

```
$KI
$AS DT:,1
$RU X
$RU Y
```

During programs X and Y, DECtape unit Ø is treated as logical unit 1.

b. When a program is in core, assignments made through an ASSIGN command are valid only through the running of that program. For example:

```
$GE X
$AS DT:,1
 .
 .
 .
$KI
```

During the run of program X, DECtape unit Ø is treated as logical unit 1. The KILL command returns the device assignments to their original default conditions.

c. When a program is in core and executing, an assignment made by an ASSIGN command is valid only through the running of that program. For example:

```
$RU X
PLEASE ASSIGN INPUT ON UNIT 4
AØØ5  1ØØ                     ←--------(PAUSE message)
$AS  SY:FILE1.XX,4
$CO
```

During the run of program X, the FORTRAN program wrote a message requesting a device assignment and then executed a PAUSE statement. The user then indicated that the desired file was to be read from the system disk. Following the execution of program X, the device table reverts to its default assignments.

11.5 THE FORTRAN DEVICE TABLE, $DEVTB

Logical device assignment is governed by the device table, which contains entries for eight devices. Additional entries can be inserted, default entries changed, or entries deleted by reassembling the device table file. To allow for such changes, the source file for the device table (DVB.MAC) is supplied with each FORTRAN system.

When the source has been altered and assembled, the new device table object module can replace the supplied module in the FORTRAN Library or be linked as an object module (to individual programs) preceding the library file. The device table consists of a 12-word header followed by a 16-word entry for each device.

Table 7-17

Device Table Entry

	Header	
	Word Ø	Address of device table entry for logical unit -3 (the message logging unit)
	Word 1	Address of entry for error routine message file
$DEVTB:	Word 2	Number of entries in device vector table
	Word 3	Device number of message logging file
	Words 4-11	Addresses of device table entries for each of the devices 1 through 8
	Entry	
	Word 1	Link pointer (from Link Block, after .INIT)
	Word 2	Physical device name stored in Radix-5Ø format
	Word 3	Bits 15-8 = unit number (Default Ø); Bits 7-Ø = DOS Open Code, taken from File Block-2
	Words 4-5	Filename (stored as .RAD5Ø/FOR/,/NNN/; NNN = Entry number)
	Word 6	Filename extension (stored as .RAD5Ø/DAT/)
	Word 7	Switches* and protection code (Default = 233)
	Word 8	Status/Mode (from Line Buffer Header)
	Word 9	Count of I/O operations for this device
	Words 1Ø-14	Unused for formatted and unformatted I/O
	Word 15	User Identification Code (UIC) (default=Ø, indicating the current UIC)
	Word 16	Address of error variable (IERR from CALL SETFIL)

Table 7-17 (Cont.)

For Random I/O, Words 8-14 are as follows:

Word 8	Function word
Word 9	Block number
Word 1Ø	Buffer address
Word 11	Buffer length
Word 12	Associated variable address (from DEFINE FILE)
Word 13	Maximum number of records (from DEFINE FILE)
Word 14	Record length (from DEFINE FILE)

*Switches are as follows:

Bit	Setting	Meaning
Ø-1	Ø	Closed
	1	Open formatted
	2	Open unformatted
	3	Open random
2	1	DEFINE FILE done on this device

Table 7-17 describes the device table header and individual entry formats. The supplied device table is listed in Chapter 7-17.

In order to add a device to the table,

1. Alter the word at $DEVTB to reflect the number of entries in the device table (the number of devices available to FORTRAN). The supplied value at $DEVTB is 8.
2. Append the address of the device entry to the 8-word header sequence of device table entry addresses (known as the device table entry vector).
3. Append the new 16-word entry describing the additional device.

To delete a device from the table,

1. Alter the word at $DEVTB to reflect the entries in the device table.
2. Set the address of the device entry in the device table entry vector to zero.
3. Delete the entry for that device from the device table.

The default physical device for a table entry can be changed by modifying the second word of the entry. This allows the user to alter the associations between devices and logical unit numbers shown in Table 7-17.

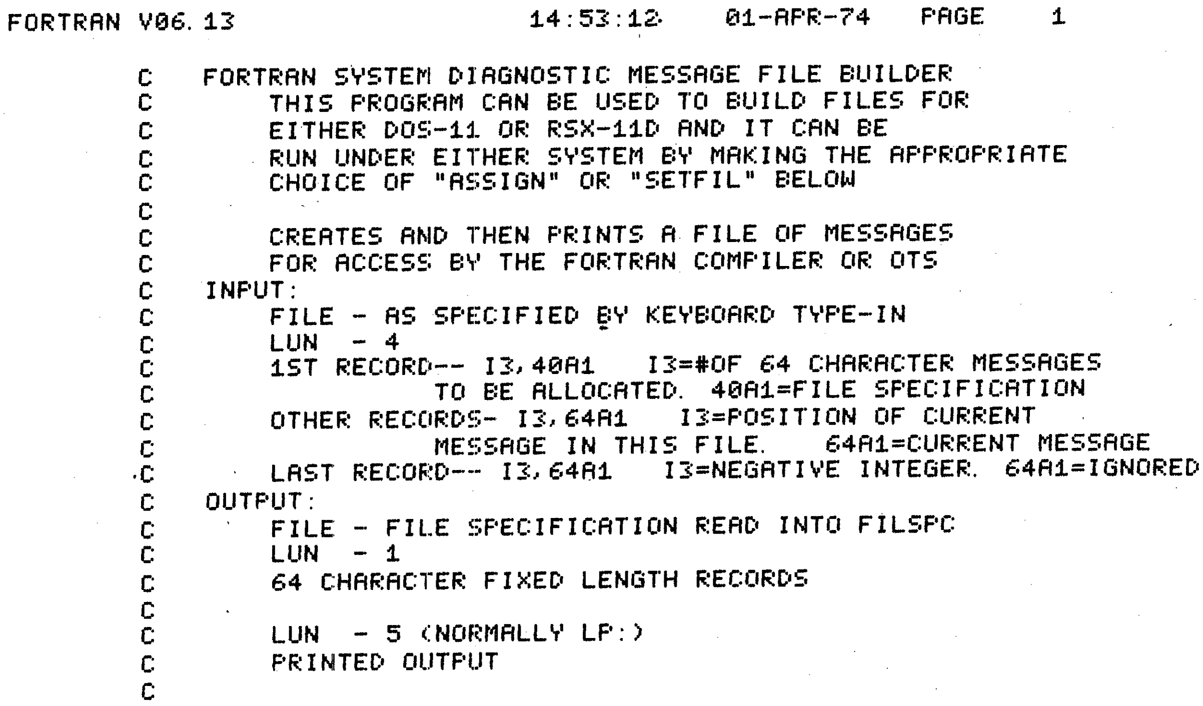
The default filename and extension for a table entry can be changed by modifying words 4 through 6 of the table entry.

11.6 I/O EXAMPLE

This section presents an example using the FORTRAN I/O system. The program FORDGN.-FTN on the following pages is used in the FORTRAN system to read an ASCII source file and produce a contiguous file of specially formatted records containing the English language diagnostic messages for the FORTRAN Compiler and OTS. A listing of the message file is also printed on the line printer in the example following.

The program reads an input file from logical unit 4. The first record of the input file specifies the number of error messages in the output file and the name of the output file. CALL SETFIL sets the name of the output file and the DEFINE FILE statement specifies the record structure.

The source listing of FORDGN.FTN follows.



```
FORTRAN VØ6.13                  14:53:12    Ø1-APR-74   PAGE    1

        C   FORTRAN SYSTEM DIAGNOSTIC MESSAGE FILE BUILDER
        C       THIS PROGRAM CAN BE USED TO BUILD FILES FOR
        C       EITHER DOS-11 OR RSX-11D AND IT CAN BE
        C       RUN UNDER EITHER SYSTEM BY MAKING THE APPROPRIATE
        C       CHOICE OF "ASSIGN" OR "SETFIL" BELOW
        C
        C       CREATES AND THEN PRINTS A FILE OF MESSAGES
        C       FOR ACCESS BY THE FORTRAN COMPILER OR OTS
        C   INPUT:
        C       FILE - AS SPECIFIED BY KEYBOARD TYPE-IN
        C       LUN  - 4
        C       1ST RECORD-- I3,4ØA1    I3=#OF 64 CHARACTER MESSAGES
        C                 TO BE ALLOCATED. 4ØA1=FILE SPECIFICATION
        C       OTHER RECORDS- I3,64A1    I3=POSITION OF CURRENT
        C                 MESSAGE IN THIS FILE.    64A1=CURRENT MESSAGE
        C       LAST RECORD-- I3,64A1    I3=NEGATIVE INTEGER. 64A1=IGNORED
        C   OUTPUT:
        C       FILE - FILE SPECIFICATION READ INTO FILSPC
        C       LUN  - 1
        C       64 CHARACTER FIXED LENGTH RECORDS
        C
        C       LUN  - 5 (NORMALLY LP:)
        C       PRINTED OUTPUT
        C
```

```
              INTEGER COUNT,INDEX
              BYTE TODAY(9),FILSPC(40),A(64),NULL(64)
              DATA NULL / 64 * 0 /
              COUNT=0
              WRITE (6,1000)
      1000    FORMAT('$SPECIFY INPUT FILE>')
              READ(6,10001) I0,FILSPC
      10001   FORMAT(Q,40A1)
      C
      C       SET UP THE INPUT FILE NAME ASSIGNMENT
      C
      C       CALL ASSIGN(4,FILSPC,I0)
              CALL SETFIL(4,FILSPC,IERR,'SY',0)
      C
              ENDFILE 6
      C
      C       READ FIRST RECORD AND ALLOCATE CONTIGUOUS FILE
              READ(4,1001) IBLOK,FILSPC
      1001    FORMAT(I3,40A1)
      C
      C       SET UP THE OUTPUT FILE NAME ASSIGNMENT
      C
      C       CALL ASSIGN(1,FILSPC,40)
              CALL SETFIL(1,FILSPC,IERR,'SY',0)
      C
              DEFINE FILE 1(IBLOK, 32, U, INDEX)
      C
      C       INITIALIZE ALL RECORDS OF OUTPUT FILE TO ZEROS
              DO 10 I=1,IBLOK
      10      WRITE (1'I)NULL
      C

FORTRAN V06.13                  14;53:12    01-APR-74   PAGE    2

      C       READ INPUT AND WRITE EACH RECORD OF OUTPUT FILE
      40      READ (4,1002,END=50)I0,I1,A
      1002    FORMAT(Q,I3,64A1)
              IF (I1 .LT. 0) GO TO 50
              COUNT=COUNT+1
      C
              I0=I0-3
              IF (I0 .LE. 0) GO TO 40
              IF (I0 .LT. 64)A(I0+1)=0
      C
              I1=I1+1
              WRITE (1'I1)A
              GOTO 40
      C
      C       READ THE CONTIGUOUS FILE AND
      C       PRINT A LISTING OF THE COMPLETED FILE
      50      ENDFILE 4
              CALL DATE(TODAY)
              WRITE (5,1003) FILSPC, TODAY
      1003    FORMAT(1H1,28X,40A1/1H0,'MSG MESSAGE',17X,9A1/' NUM'/)
      C
              DO 51 I1=1,IBLOK
              READ(1'I1)A
              I2=I1-1
      51      WRITE (5,1004) I2,A
      1004    FORMAT(X,I3,X,64A1)
              WRITE (5,1005) IBLOK,COUNT
      1005    FORMAT(/X,I3,' MESSAGES ALLOCATED.',I6,' MESSAGES INPUT.')
              ENDFILE 1
              ENDFILE 5
              END
```

```
          ROUTINES CALLED:
          SETFIL, DATE

          OPTIONS =/OP:1

          BLOCK        LENGTH
          MAIN.    584    (002220)*

          **COMPILER ----- CORE**
             PHASE         USED  FREE
          DECLARATIVES 00622 06070
          EXECUTABLES  00863 05829
          ASSEMBLY     01355 09977
```

The input file, ABC.SRC:

```
018ABC.TST
002  FORT000002 SUBROUTINE DIRECTLY (INDIRECTLY)REFERENCES ITSELF
003  FORT000003 ILLEGAL FLOATING POINT INSTRUCION
004  FORT000004 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
000  FORT000000 INVALID CALL TO ERROR
001  FORT000001 NO SPACE TO DO I/O
005  FORT000005 SYSTEM ERROR NO DIAGNOSTIC ASSIGNED
006  FORT001000 SYSTEM ERROR NO DIAGNOSTIC ASSIGNED
007  FORT001001 DEVICE PARITY ERROR
008  FORT001002 CHECKSUM/PARITY ERROR-END OF DATA ERROR (RANDOM)
010  FORT001004 END OF FILE OR END OF MEDIUM
012  FORT001006 DEFINE FILE NOT DONE (RANDOM ACCESS)
013  FORT001007 DEFINE FILE DONE (NOT RANDOM ACCESS)
015  FORT001009 FILE DOES NOT EXIST / OR IS ALREADY OPEN
016  FORT001010 UNABLE TO OPEN FILE
```

Before execution of the program, the disk contains the following.

```
#/DI

DIRECTORY DK0: [ 50,51 ]

01-APR-74

FORDGN.FTN      6  01-APR-74 <233>
ABC   .BAK      2  01-APR-74 <233>
FORDGN.OBJ      8  01-APR-74 <233>
FORDGN.LDA     20  01-APR-74 <233>
ABC   .SRC      2  01-APR-74 <233>

TOTL BLKS:     38
TOTL FILES:     5
```

The program is run and the output file ABC.TST is created.

```
    $ASSIGN SY:ABC.SRC,4
    $RUN FORDGN.LDA

    SPECIFY INPUT FILE>ABC.SRC

    $RUN PIP
    PIP  V10-02
    #LP:</DI

    #


    DIRECTORY DK0: [ 50,51 ]

    03-APR-74

    FORDGN.FTN      6  01-APR-74 <233>
    ABC   .BAK      2  01-APR-74 <233>
    FORDGN.OBJ      8  01-APR-74 <233>
    FORDGN.LDA     20  01-APR-74 <233>
    ABC   .SRC      2  01-APR-74 <233>
    ABC   .TST     3C  01-APR-74 <233>

    TOTL BLKS:     41
    TOTL FILES:     6
```

The listing of the file produced on the line printer:

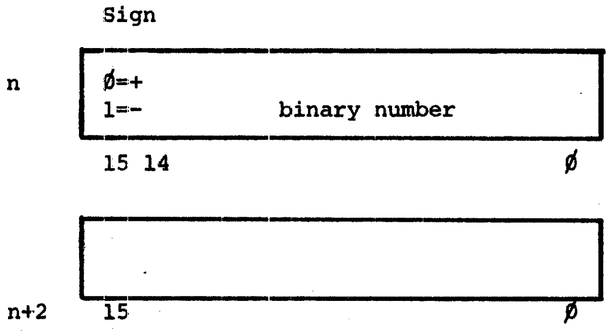
```
                          ABC.TST

MSG MESSAGE               01-APR-74
NUM

  0   FORT000000 INVALID CALL TO ERROR
  1   FORT000001 NO SPACE TO DO I/O
  2   FORT000002 SUBROUTINE DIRECTLY (INDIRECTLY)REFERENCES ITSELF
  3   FORT000003 ILLEGAL FLOATING POINT INSTRUCION
  4   FORT000004 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
  5   FORT000005 SYSTEM ERROR NO DIAGNOSTIC ASSIGNED
  6   FORT001000 SYSTEM ERROR NO DIAGNOSTIC ASSIGNED
  7   FORT001001 DEVICE PARITY ERROR
  8   FORT001002 CHECKSUM/PARITY ERROR-END OF DATA ERROR (RANDOM)
  9
 10   FORT001004 END OF FILE OR END OF MEDIUM
 11
 12   FORT001006 DEFINE FILE NOT DONE (RANDOM ACCESS)
 13   FORT001007 DEFINE FILE DONE (NOT RANDOM ACCESS)
 14
 15   FORT001009 FILE DOES NOT EXIST / OR IS ALREADY OPEN
 16   FORT001010 UNABLE TO OPEN FILE
 17

 18 MESSAGES ALLOCATED.    14 MESSAGES INPUT.
$
```

PART 7

CHAPTER 12

FORTRAN WORD FORMATS

12.1 INTEGER FORMAT



In a 2-word format, an integer is assigned two storage words, although only the high-order word (i.e., the word having the lower address) is significant. Use of the /ON switch (see Appendix J) causes integers to be assigned a single storage word. Negative integers are stored in a two's complement representation. Integer data must lie in the range -32768 to +32767, For example:

+22 = ØØØ026$_8$
-7 = 177771$_8$

12.2 FLOATING-POINT FORMATS

All floating-point data, in both 2-word and 4-word formats, is stored as a 1-bit sign, an 8-bit exponent (characteristic), and a fractional mantissa. The exponent is stored in excess 128 notation. That is, a binary zero represents an exponent of -128; the binary equivalent of 255 (377$_8$) represents an exponent of +127. The mantissa is stored as a binary fraction; that is, the most significant bit represents Ø.5, the next bit represents Ø.25, the third, Ø.125, and so on. Because all numbers are presumed to be normalized, the most significant bit is redundant and is therefore not stored (this is called hidden bit normalization). If the exponent is zero (corresponding to 2^{-128}), the most significant bit is assumed to be Ø; for all other exponent values, it is assumed to be 1. A value of zero is represented by two or four words completely filled with zeros. As an example of floating-point storage, the value +1.Ø is represented in the 2-word format by

```
Ø4Ø2ØØ
ØØØØØØ
```

or in the 4-word format by the following.

```
Ø4Ø2ØØ
ØØØØØØ
ØØØØØØ
ØØØØØØ
```

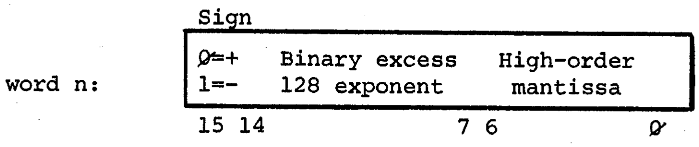
This is equivalent to an exponent of 129 (corresponding to 2^1) multiplied by a mantissa of zero (assumed to have a value of Ø.5). The value -5.0 in the 2-word format is

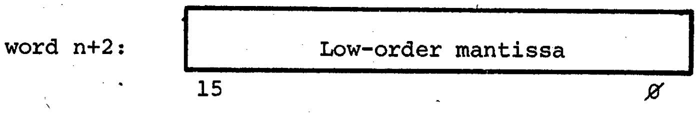
```
14Ø64Ø
ØØØØØØ
```

or in the 4-word format, the following.

```
14Ø64Ø
ØØØØØØ
ØØØØØØ
ØØØØØØ
```

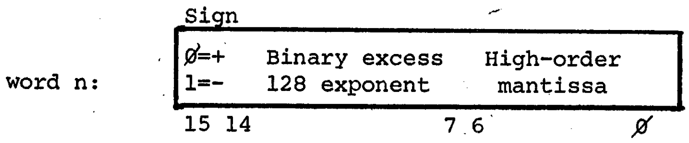
12.2.1 Real Format (2-Word Floating-Point)

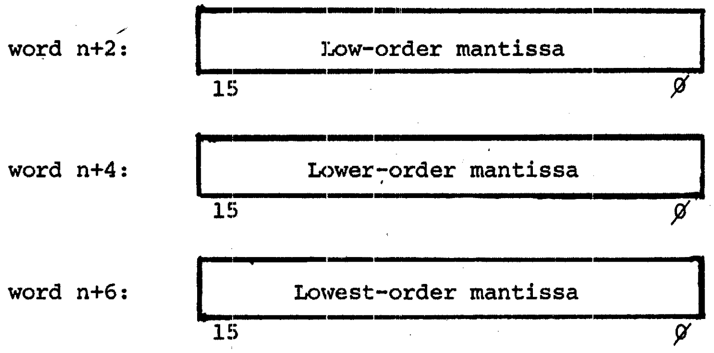




Real floating-point format gives an effective precision of 24 bits, or approximately 7 digits of accuracy. The magnitude range lies between approximately Ø.14 x $1Ø^{-38}$ and 1.7 x $1Ø^{38}$.

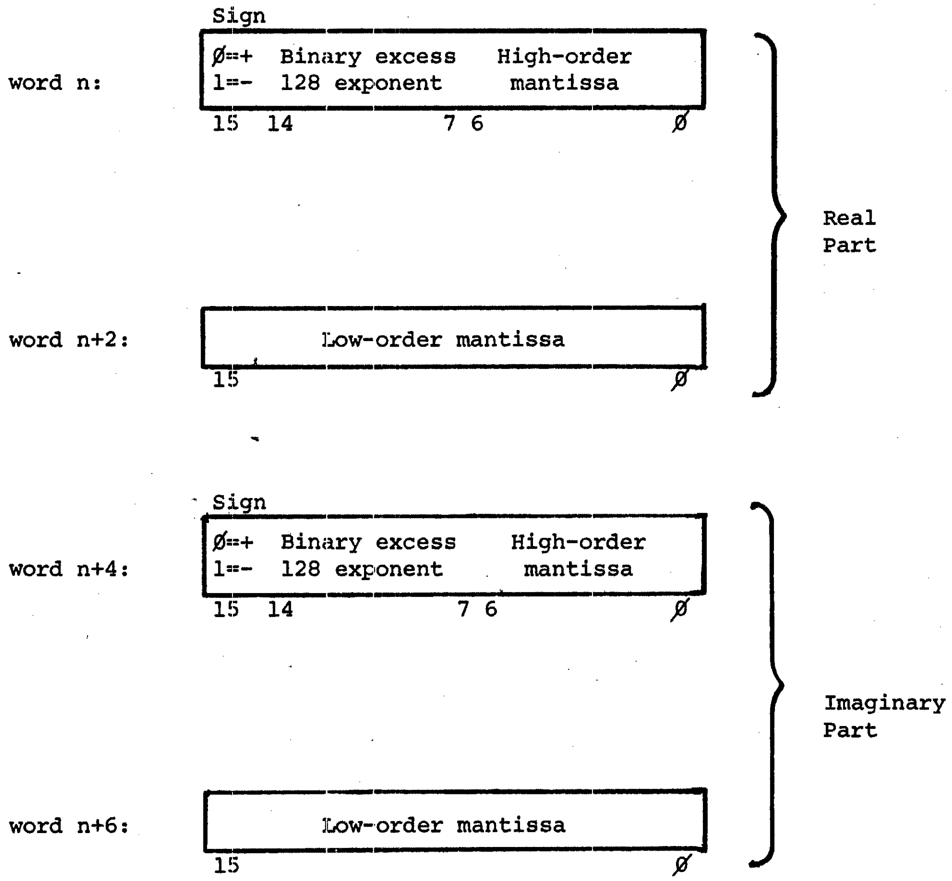
12.2.2 Double Precision Format (4-Word Floating-Point)



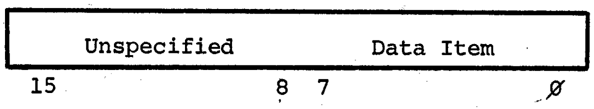


The effective precision is 56 bits, or approximately 16 decimal digits, or accuracy. The magnitude range lies between ∅.14 x $1∅^{-38}$ and 1.7 x $1∅^{38}$.

12.3 COMPLEX FORMAT

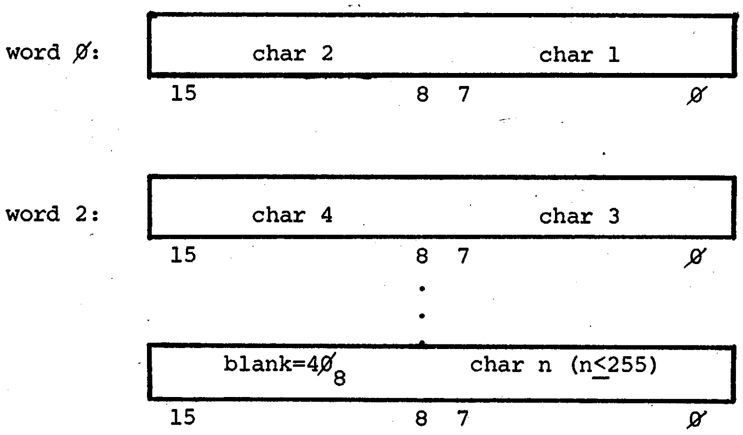


12.4 BYTE FORMAT



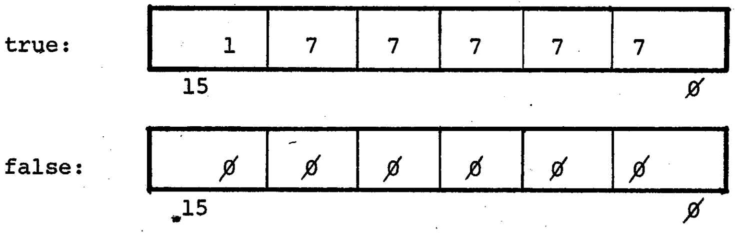
The range of numbers from +127 to -128 can be represented in BYTE (LOGICAL *1) format. BYTE format array elements are stored in adjacent bytes.

12.5 HOLLERITH FORMAT



Hollerith constants are stored internally as one character per byte, filling up to word boundaries. An odd number of characters causes a blank to be appended to the constant. A Hollerith constant included in an arithmetic expression is treated as a 1-word integer quantity where only the first two characters of the constant are significant.

12.6 LOGICAL FORMAT



Logical format data items are treated as 1-word integer values for use with arithmetic and logical operators. Any nonzero value is considered to have a logical value of .TRUE. when tested by a logical IF statement.

12.7 RADIX-5Ø FORMAT

See Appendix B.

PART 7

CHAPTER 13

FORTRAN LIBRARY FUNCTIONS

This Chapter contains a brief outline of the OTS library of FORTRAN functions that involve approximations. Floating-point means single-precision, 2-word, floating-point format with a 24-bit fraction and an 8-bit binary exponent. Double-precision means 4-word, floating-point format with a 56-bit fraction and an 8-bit binary exponent. The values of the coefficients used in the various approximations may be found at the cited parts of the following references.

a. Computer Approximations, by J.F. Hart et al, John Wiley & Sons, 1968.

b. Approximation for Digital Computers, by C. Hastings et al, Princeton University Press, 1955.

c. PDP-11 Paper Tape Software Programming Handbook, DEC-11-GGPC-D, Digital Equipment Corporation.

In the descriptions of the various functions, the relative error values given are for the approximating polynomials in the cited intervals. These error bounds assume exact arithmetic. There are two additional sources of errors in the function calculations that are not considered in the error bounds.

a. Rounding and truncation errors can occur in reducing the given argument to the range in which the polynomial or rational fraction approximations are valid.

b. Rounding errors can occur as a result of using finite precision floating-point arithmetic in the polynomial or rational fraction computations.

All OTS FORTRAN functions are called using one of the following standard sequences (see Chapter 7-15).

```
                                                  MOV      #LIST,RE
        JSR      R5,NAME          or              JSR      PC,NAME
LIST:   BR       RTN                      LIST:   BR       RTN
        .WORD    ARG1,...,ARGn                    .WORD    ARG1,...,ARGn
RTN:                                      RTN:
```

The result is returned in RØ-R1 for floating-point and RØ-R3 for a double-precision function.

Some FORTRAN functions call other single argument functions in the course of their computation. In order for them to be re-entrant, these calls are made via the routine $FCALL.

```
MOV     ARGUMENT ADDRESS, R5
MOV     #FUNCTION NAME, R4
JSR     PC,$FCALL
```

$FCALL calls the FORTRAN function whose address is in R4 with the argument whose address is in R5. Control is returned to the instruction following the JSR with the function result in RØ-R1 for floating-point or RØ-R3 for double-precision.

Here are the FORTRAN Library approximation functions.

1. ALOG(X), Floating-point Natural Logarithm

   If $X<Ø$ call error

   Let $X=y*2^a$ where $1/2\leq y<1$

   Let $Q=(y*\sqrt{2}-1)/(y*\sqrt{2}+1)$

   Then $\ln(X) = a * \ln(2) + \ln(y)$

   $ALOG(X) = a*\ln(2) = \ln(\sqrt{2}) + Q\Sigma_{Ø}^{3} c_i Q^{2i}$

   where the c_i are drawn from Hart #2662. The relative error is $\leq 1Ø^{-9.9}$.

2. ALOG1Ø(X), Floating point Common Logarithm

   Computed as $\log_{1Ø}(e)$*ALOG(X).

3. ATAN(X), Floating-point Arctangent

   If $X<Ø$, ATAN(X) = -ATAN (-X)

   If $|X|>1$, $ATAN(|X|) = \pi/2 - ATAN\ (1/|X|)$

   If $|X|>\tan \pi/12$, $ATAN(X) = \pi/6 + ATAN\ ((X\sqrt{3}-1)/(X+\sqrt{3}))$

   For $|X|\leq\tan \pi/12$, $ATAN(X) = X\Sigma_{Ø}^{4}c_i*X^{2i}$

   where the c_i are drawn from Hart #4941. The relative error is $\leq 1Ø^{-9.5}$.

4. ATAN2(X,Y),Two Argument Floating-point Arctangent

   If $Y = Ø$, or $X/Y >2^{25}$, ATAN2(X,Y) = $\pi/2$ (sign X).

   If $Y > Ø$, and $X/Y\leq 2^{25}$, ATAN2(X,Y) = ATAN(X/Y).

   If $Y < Ø$, and $X/Y\leq 2^{25}$, ATAN2(X,Y) = π*signX + ATAN(X/Y).

5. DATAN(X), Double-precision Arctangent

   The analysis is the same as in that for ATAN(X) except that the polynomial approximent is of degree 8. The coefficients are drawn from Hart #4945. The relative error is $\leq 1Ø^{-16.8}$.

6. DATAN2(X,Y), Two Argument Double-precision Arctangent

The rules for DATAN2 are the same as those for ATAN2 except that the DATAN is used in all computations.

7. DLOG(X), Double-precision Natural Logarithm

The analysis for DLOG is the same as that for ALOG except that the polynomial in Q^2 is of degree 6. The c_i are drawn from Hart #2665. The relative error is $\leq 10^{-16.5}$.

8. DLOG10(X), Double-precision Common Logarithm

Computed as $\log_{10}(e)$*DLOG(X).

9. DSQRT(X), Double-precision Square Root

If X<0 call error

Let $X = A*2^B$ where $1/2 \leq A < 1$

Let $Y_0 = 2^{B/2} * (1/2 + A/2)$ if B is even

or

$Y_0 = 2^{(B+1)/2} * (1/4 + A/2)$ if B is odd

a transformation requiring only two instructions. Starting with Y_0, four Newton-Raphson iterations are performed.

$$Y_{n+1} = 1/2(y_n + x/y_n).$$

The relative error is $<10^{-17}$.

10. DSIN(X), Double-precision Sine

Let y = Integer (4 * fraction (X/2π))

Let V = Fraction (4 *fraction (X/2π))

$$\begin{aligned} \text{Then DSIN(X)} &= P(V\pi/2) \text{ if } y=0 \\ &= P((1-V)\pi/2) \text{ if } y=1 \\ &= P(-V\pi/2) \text{ if } y=2 \\ &= P((V-1)\pi/2) \text{ if } y=3 \end{aligned}$$

where $\sin(V\pi/2) \approx P(V\pi/2) = V\sum_{0}^{8} c_i V^{2i}$ for $-1 \leq V \leq 1$

The c_i are drawn from Hart #3345. The relative error is $\leq 10^{-18.6}$.

11. DCOS(X), Double-precision Cosine

Computed as DSIN(X + π/2).

12. DEXP(X), Double-precision Exponential

If X > 87 call overflow

If $|X| < 2^{-60}$, DEXP(X) = 1

If X < -88.7, DEXP(X) = 0

Let y = Integer $(X*\log_2(e))$

Let V = 16 * Fraction (X · $\log_2(e)$)

Let w = 1/16 * Fraction (V)

DEXP = $2^Y * 2^w * 2^{\text{Integer (V)/16}}$ where Ø $\leq$ w < 1/16.

Powers of $2^{1/16}$ are obtained from a table.

$2^w \approx \frac{P(w^2)+wQ(w^2)}{P(w^2)-wQ(w^2)}$ where P and Q are first degree polynomials in w^2.

The coefficients of P and Q are drawn from Hart #1121.

The relative error is $\leq$1Ø$^{-16.4}$.

13. EXP(X), Floating-point Exponential

If X > 87, call overflow

If $|X| < 2^{-28}$, EXP(X) = 1.

If X < -88.7, (EXP(X) = Ø.

Let y = Integer (X * $\log_2(e)$)

Let V = Fraction (X *$\log_2(e)$)

Let w = 1/2 ln(2) * V where $|X| \leq \ln(2)/2$

Then EXP(X) = $2^Y * (e^w)^2$

where $e^w \approx 1 + \cfrac{2 * w}{c_1 - w - \cfrac{c_2}{c_3 + w^2}}$

The c_i are drawn from DEC-11-GGPC-D, page 7-23. The relative error is $\leq$1Ø$^{-1Ø}$.

14. RAN(I1,I2) and CALL RANDU(I1,I2,F), Random Number Generator

RAN and RANDU use the same multiplicative congruential algorithm for generating uniformly distributed pseudo-random numbers in the interval (Ø,1).

If I1=Ø, I2=Ø set generator base

$$X_{n+1} = 2^{16}+3$$

otherwise

$$X_{n+1} = (2^{16}+3) \quad X_n \bmod 2^{32}$$

Store generator base X_{n+1} in I1,I2.

Result is X_{n+1} scaled to a real value Y_{n+1}, Ø<Y_{n+1}<1.

15. SIN(X), Floating-point Sine

The analysis is the same as that for DSIN(X). The polynomial approximant used is of degree 4 and the coefficients are drawn from Hastings, sheet 16. The relative error is $\leq 2 * 10^{-8}$.

16. COS(X), Floating-point Cosine

Computed as SIN(X + π/2).

17. SQRT(X), Floating-point Square Root

The analysis is the same as that for DSQRT(X) except that only three iterations are performed. The relative error is $\leq 10^{-8}$.

18. TANH(X), Floating point Hyperbolic Tangent

If $|X| \geq 16$, TANH(X) $\approx$ 1 * sign(X)

otherwise

let y = EXP(2 * X)

TANH(X) = (y-1)/(y+1).

PART 7

CHAPTER 14

SYSTEM SUBPROGRAMS

14.1 LIBRARY ARITHMETIC FUNCTIONS

Refer to Table 7-3 for the FORTRAN Library arithmetic functions.

14.2 SYSTEM SUBROUTINES AND FUNCTIONS

The FORTRAN library contains subroutines that may be called in the same manner as user-written subroutines, similiar to the functions intrinsic to the FORTRAN system. These subroutines appear in Table 7-18.

Table 7-18

FORTRAN Library Subroutines

System Subroutine	Function
PDUMP	performs core dumps of specified sections of core.
SETPDU	changes logical unit to which PDUMP output is written.
SETFIL	overrides default value for a logical unit at run time.
ASSIGN	allows specification at run-time of filename or device and filename to be associated with a FORTRAN logical unit number.
SETERR	allows the user to reset the maximum occurrence count for any class of error except Ø.
TSTERR	returns an indication of whether an error of the specified class has occurred.
RANDU,RAN	returns a single random number with uniform distribution between Ø and 1.
EXIT	terminates the execution of a program and returns control to the Monitor (also invoked automatically by the error processing routines and STOP statement).
DATE	returns a 9-byte string containing the ASCII representation of the current date.
TIME	returns an 8-byte string containing the ASCII representation of the current time in hours, minutes and seconds.

Table 7-18 (Cont)

FORTRAN Library Subroutines

System Subroutine	Function
SECNDS	provides system time of day or elapsed time as a floating-point value in seconds.
IRAD5Ø RAD5Ø	performs conversion of Hollerith strings to Radix-5Ø representation.
R5ØASC	converts Radix-5Ø strings to Hollerith strings.
SSWTCH	tests specified bits of the console switch register.

14.2.1 PDUMP

The PDUMP subroutine causes specified portions of core to be dumped. Control returns to the calling program following execution of the dump. The call to PDUMP is performed as follows:

CALL PDUMP $(L_1,U_1,F_1,\ldots,L_n,U_n,F_n)$

where L_1 is the lower limit (a variable name) and U_1 is the upper limit of the area to be dumped. The order in which the two variables appear may be reversed; regardless of the order of specification, memory is always dumped from the lower limit to the upper limit, inclusive. For example,

```
DIMENSION N(1Ø)
    .
    .
    .
CALL PDUMP (N(1),N(1Ø),3)
```

causes the array N to be dumped on the logical unit 5 (generally LP:) as a string of 2-word integers from N(1) to N(1Ø). The following line performs exactly the same dump.

```
CALL PDUMP (N(1Ø),N(1),3)
```

The value of F_i is an integer constant or variable indicating the dump format, as follows.

Code	Description	Format
Ø	1-word octal	8O8
1	1-word decimal	8I8
2	2-word octal	8O8
3	2-word decimal	8I8
4	real	4G15.7
5	double precision	3D22.14

Codes Ø and 1 should be used if the /ON (1-word integer) switch is set; codes 2 and 3 should be used if the /ON switch is not set. Codes Ø and 1 print every word while codes 2 and 3 print every other word, since the alternate words are unused. No default dump mode is assumed; a code must be specified or a diagnostic will be generated. Where several sections of core are to be dumped, each set of three values (L_i,U_i,F_i) specifies a separate core area and dump format.

No spacing is performed between dumps of individual sections, nor are addresses printed on the PDUMP output.

As an example, the output shown below is produced using the call:

```
     CALL PDUMP (B(3),B(3),Ø,R(1),R(7),Ø,F(1),F(4),5,S(1),S(24),Ø)
```

where the program contains:

```
          DOUBLE PRECISION F(4)
          INTEGER B(3),R(7),S(24)
```

and the /ON (one-word integer) switch is used at compilation time.

```
     1404       14     12746
        Ø        Ø         Ø       Ø   2Ø124   2Ø13Ø  17776
                        Ø.  Ø.197215245Ø87Ø1D-3Ø Ø.986Ø762431Ø639D-31
Ø.493Ø3812265972D-31
 22Ø6Ø   3477Ø   17756         5ØØØ2       3Ø  21172    4Ø6    2Ø13Ø
     Ø    4467   12672         3224Ø        Ø      Ø  5Ø561    5574Ø
 327Ø2       1   2ØØ24          4567     1412    4ØØ   4467    1264Ø
```

14.2.2 SETPDU

The SETPDU subroutine allows PDUMP output to be directed to any legal logical unit number. The call to SETPDU is performed as follows:

```
          CALL SETPDU (IUNIT)
```

where IUNIT is a constant, variable or expression of type INTEGER designating the logical unit desired.

When used in conjunction with the ASSIGN subroutine (see Section 7-15.2.4), PDUMP output may be directed to any file or device on the system.

14.2.3 SETFIL

The SETFIL subroutine overrides the default value for a logical device assignment at run time. The SETFIL call must occur before the file in question has been opened for I/O operations (by a READ or WRITE statement). The new logical device assignment remains in effect until the end of the current program run or until the file is closed by END FILE and a new device assignment made. The call to SETFIL appears as shown below:

```
          CALL SETFIL (N,FILE,IERR,DEV,NU,UIC,PC,CS,LREC,NREC)
```

SETFIL accepts a variable number of arguments; nonessential arguments may be omitted. The complete list of arguments to this call appears in Table 7-19.

Table 7-19

FORTRAN SETFIL Arguments

Argument	Significance
N	Logical unit number, expressed as an integer value.
FILE	Character string containing the file name and extension specification, of the form: file.ext where "file" may be up to six alphanumeric characters, of which the first is alphabetic. Optionally, ".ext" may be specified to indicate an extension. The entire file specification is an ASCII string that can be expressed as a Hollerith string or enclosed in single quotes. For example:

Table 7-19

FORTRAN SETFIL Arguments (Cont.)

Argument	Significance
	'MAT.TMP' 8HOBJ2.DAT See also the discussion of string argument storage in Section F.2.13.
IERR	Integer variable into which the error returns from this routine or from I/O operations on logical unit "N" are placed. IERR is set to -1 if CS is neither 1 nor 2; it is set to 2 if CS=1 and neither LREC nor NREC is specified.
DEV	A 2- or 3-character device specification, expressed as a Hollerith string or enclosed in single quotes. For example: 'DT' 2HLP
NU	Unit number of the device specified, if appropriate. This is an integer constant or variable (e.g., 1 if the device is DECtape unit 1, or Ø if a line printer or other single-unit device).
UIC	User identification code (UIC), expressed as an integer value. A UIC specification of Ø indicates the current UIC (the usual specification). A simple way of preparing the UIC entry is shown below (assume the desired UIC is 12Ø,15Ø) INTEGER UIC LOGICAL*1 UICB(2) EQUIVALENCE (UICB(1),UIC) UICB(1) = "15Ø UICB(2) = "12Ø . . . CALL SETFIL (2,'DATA.TMP',IERR,'DT',2,UIC) which specifies that the file DT2:DATA.TMP [12Ø,15Ø] is to be accessed on logical device 2.
PC	Protection code, specified as an integer value, which is the decimal equivalent of the desired octal protection code (protection codes are defined with octal values in Part 3 of this handbook). The default value for a FORTRAN protection code is 233(8).
CS	Integer value used to indicate one of the following: CS = 2, allocate a contiguous file for random I/O. The DEFINE FILE statement sets the record length and number of records in the file. CS = 1, allocate a contiguous file for formatted or unformatted I/O. The arguments LREC and NREC are used to set the record length and number of records in the file.
LREC	Logical record length in words, expressed as an integer value; required if CS=1, ignored otherwise.
NREC	Number of records to allocate, expressed as an integer value; required if CS=1, otherwise ignored.

The discussion of file formats in Section 7-11.2 should be consulted as a guide in choosing values for LREC and NREC.

When SETFIL is used in an overlay system, a special restriction exists for the variable IERR. In the statement

```
          CALL SETFIL (7,'FILE.DAT',IERR,'DK',Ø)
```

the address of the variable IERR is saved and is used by the I/O system for reporting certain kinds of I/O error conditions. This variable name is saved by SETFIL in the device table, $DEVTB. Any subsequent I/O errors (during READ or WRITE operations) attempt to set IERR. Therefore, IERR must be in the local storage of the resident section or in a resident common block. If it is not, the FORTRAN I/O system may destroy an unpredictable location in memory when attempting to store error data in IERR. Further discussion of the use of SETFIL and of the FORTRAN I/O system is presented in Chapter 7-11. Two examples of the use of SETFIL are given below. These examples assume the default logical unit assignments.

Example 1: Make logical unit 5 the card reader and logical unit 6 the line printer.

```
            .
            .
            .
            CALL SETFIL (5,'A',IERR,'CR')
            CALL SETFIL (6,'A',IERR,'LP')
            .
            .
            .
```

Note that the parameters to SETFIL are positional. The CR: and LP: are not file structured, so the file name argument is superfluous; a file name argument must be included, however.

Example 2: Allocate a file for direct-access I/O on disk unit 1 under the current UIC.

```
            .
            .
            .
            CALL SETFIL (1,'FILE1.XYZ',IERR,'SY',Ø,Ø,1ØØ,2)
            DEFINE FILE 1 (2Ø,2ØØ,U,INDX1)
            .
            .
            .
```

The file FILE1.XYZ is created on the system disk with protection code 1ØØ (read, write, execute access; the file cannot be deleted until its protection code is changed).

14.2.4 ASSIGN[1]

The ASSIGN subroutine assigns, at run time, the device and/or file name to be associated with a logical unit number. The ASSIGN call must occur before the logical unit is opened for I/O operations (by READ or WRITE). The assignment remains in effect until the end of the program or until the file is closed by END FILE and a new CALL ASSIGN is performed. The call to ASSIGN is performed as follows.

```
          CALL ASSIGN (N,NAME,ICNT,IERR)
```

The arguments to this routine follow.

Argument — Significance

N — Logical unit number, expresses as an integer value.

NAME — Character string containing any device and file name acceptable to the Monitor system.

The maximum acceptable length for a name string is 3Ø characters.

If the device is not specified, the device remains unchanged from that specified by default or by a previous ASSIGN or SETFIL call. If the device is specified, the filename must also be specified in the case of file-structured devices. (If the device is not file-structured, a file name, if present, is ignored.)

ICNT — Actual length of the name string in characters (bytes); if the value is Ø, then the name string itself will be assumed to be terminated by a zero byte.

IERR — Integer variable into which error code values from this routine, or I/O operations on logical unit "N" are to be placed.

The error variable name is an optional argument and may be omitted. If specified, it should not be allocated in an overlay segment of the program.

14.2.5 SETERR

The system maintains an error count for each error class. The SETERR subroutine allows the user to reset the maximum occurrence count for any class of run-time error (except class Ø) to 8. (Following the maximum number of errors in a given class, control exits to the Monitor.) The call to SETERR is performed as follows.

[1] This subroutine is the preferred alternative to the SETFIL subroutine when random access file allocation is not required.

```
CALL SETERR (CLASS,MAX)
```

The argument CLASS is an integer indicating the error class affected. If CLASS is not a valid number, no action is performed. The CLASS argument is shown in relation to its associated error messages in Appendix K. The error classes appear here in brief.

Class	Default MAX Value	Meaning
Ø	-3	Fatal errors; it is recommended that the maximum occurrence count (-3) not be changed
1	Ø	Physical I/O errors
2	1	Errors in FORMAT statements
3	3	Arithmetic overflow or division by Ø
4	4	Incorrect arguments to library functions or subroutines
5	-1	Arithmetic underflow errors
6	Ø	Conversion errors
7	7	Subscripting errors

The argument MAX is an integer with the following significance:

MAX Value	Meaning
>Ø	Log message, increment error count, call EXIT if error count is > MAX.
=Ø	Log messages and ignore error count for the specified error class.
-1	Do not log messages, ignore error count for the specified error class.
-2	Do not log messages, exit to Monitor after printing an error message and closing files.
-3	Immediate exit to Monitor with a message.

14.2.6 TSTERR

The TSTERR subroutine allows the user program to monitor the types of errors detected during program execution. The call is of the form

```
CALL TSTERR(I,J)
```

where I is the error class number (between ∅ and 8); a value is returned to the variable J as follows:

```
J=1 if an error of class I has occurred.
J=2 if an error of class I has not occurred.
```

The sequence

```
      .
      .
      .
      CALL TSTERR(3,J)
      GO TO (1∅,2∅),J
   2∅ CONTINUE
      .
      .
      .
```

transfers control to statement 1∅ if a class 3 error (arithmetic overflow) has occurred. See Section 7-14.2.4 for a discussion of error classes and messages.

The TSTERR routine also resets to ∅ the error flag for that error class (but not the error count used by SETERR). For example:

```
   .
   .
   .
   CALL TSTERR(I,J)
   CALL TSTERR(I,J)
   .
   .
   .
```

The second call is guaranteed to return J=2. The TSTERR subroutine is independent of the SETERR subroutine; neither directly influences the other except that SETERR can cause execution to terminate.

14.2.7 RANDU,RAN

The random number generator may be called as a subroutine, RANDU, or as an intrinsic function, RAN. The subroutine call appears as

```
   CALL RANDU(I1,I2,X)
```

where I1 and I2 are previously-defined integer variables and X is the real variable name in which is returned a random number between ∅ and 1. I1 and I2 should be initially set to ∅. They are updated to a new generator base during each call.

Resetting I1 and I2 to Ø repeats the random number sequence. The values of I1 and I2 have a special form; only Ø or values supplied by the random number generator should be stored in these variables.

Use of the RAN function is similar to the use of the random number subroutine:

```
          RAN(I1,I2)
```

is the function reference to the random number generator.

14.2.8 EXIT

A call to the EXIT subroutine, in the form

```
          CALL EXIT
```

is equivalent to the END statement and causes program termination.

14.2.9 DATE

The DATE subroutine can be used in a FORTRAN program to obtain the current date as set within the system. The DATE subroutine is called as follows:

```
          CALL DATE (array)
```

where array is a predefined array able to contain a 9-byte string. The array specification in the call may be expressed as the array name alone,

```
          CALL DATE(A)
```

in which the first three elements of the real array A are used to hold the date string; or as

```
          CALL DATE (A(i))
```

which causes the 9-byte string to begin at the ith element of the array A.

The date is returned as a 9-byte (9-character) string in the form

```
          dd-mmm-yy
```

where:

```
          dd is the 2-digit date.
          mmm is the 3-letter month specification.
          yy is the last two digits of the year.
```

For example:

```
          27-JAN-74
```

In the case where the array is a real array, 4-1/2 words are used to contain the date string; the remaining array storage is untouched. Therefore, the date string is stored in the first nine bytes in the elements A(i), A(i+1), and A(i+2). The last three bytes of A(i+2) are untouched and should be made blank by the user if he intends to print the date with a 3A4 format.

14.2.10 TIME

The TIME subroutine allows the user to obtain the current system time or to perform time conversions within a FORTRAN program (see also SECNDS subprogram, section 7-14.2.11). The TIME subroutine may be called by any of the following three statements.

```
          CALL TIME (A)
          CALL TIME (I1,I2)
          CALL TIME (A,I1,I2)
```

The first two forms of the TIME call return the current system time; the third is used to perform time conversions.

The first type of TIME call returns the time as an 8-character ASCII string of the form:

```
          hh:mm:ss
```

where:

```
          hh is the 2-digit hour indication.
          mm is the 2-digit minute indication.
          ss is the 2-digit second indication.
```

For example:

```
          15:45:23
```

A 24-hour clock is used. A 2A4 format specification is generally used to output the time value.

The second type of TIME call returns the time in "clock ticks" (1/6∅ of a second for 60Hz systems, 1/5∅ of a second for 5∅Hz systems) elapsed since midnight, in the integer variables I(1) and I(2). I(1) contains the high-order 15 bits of the number and I(2) the low-order 15 bits. Elapsed-time calculations can be performed as follows (the use of SECNDS is recommended, however; see Section 7-14.2.11).

```
              .
              .
              .
        CALL TIME (I1.I2)
              .
              .
              .
        CALL TIME (I3,I4)
  C     COMPUTE ELAPSED TIME IN TICKS: FIRST CONVERT TICKS STORED
  C     IN I1 AND I2 TO A SINGLE FLOATING-POINT VALUE.
        START = I1*32768.∅+I2
        FINISH = I2*32768.∅+I4
  C     NOTE THAT MULTIPLICATION BY REAL CONSTANT CAUSES ENTIRE
  C     CALCULATION TO BE PERFORMED IN REAL MODE.  ELAPSED TIME
  C     IS NOT CALCULATED IN INTEGER MODE DUE TO THE POSSIBILITY
  C     OF INTEGER OVERFLOW.
        I1 = ELAPSE/32768.∅
        I2 = AMOD(ELAPSE,32768.∅)
        WRITE(6,21∅) I1,I2
  21∅   FORMAT(2O6)
              .
              .
              .
```

The above example prints the elapsed time indication as follows:

```
        ∅ 263∅
```

showing the octal representation of the number of clock ticks between the two TIME calls.

The third type of TIME call accepts I(1) and I(2) as integer values and returns the clock time as represented by I(1) and I(2) in an 8-character ASCII string in A. For example (the following is a continuation of the preceding example):

```
              .
              .
              .
        CALL TIME (A(1),I1,I2,)
  C     SCALE ELAPSED TIME TO SECONDS
        ELAPSE = ELAPSE/6∅.∅
        WRITE(6,215) ELAPSE,A(2)
  215   FORMAT('0ELAPSED TIME = !,F5.1,'SECONDS'/
       1' AS CONVERTED BY "TIME" :',2A4/)
              .
              .
              .
```

outputs the following:

```
          ELAPSED TIME = 23.9 SECONDS
          AS CONVERTED BY "TIME" : ØØ:ØØ:23
```

14.2.11 SECNDS[1]

The SECNDS function returns the system time, minus the value of the argument, as a single precision floating-point value. For example,

```
          TIM = SECNDS(Ø.)
```

will return the number of seconds since midnight; that is, the current time of day. It may be called with a non-zero argument for performing elapsed-time computations as in

```
     C    START OF TIMED SEQUENCE
          T1 = SECNDS(Ø.)
     C
     C    CODE TO BE TIMED
     C
          DELTA = SECNDS(T1)
```

where DELTA will give the elapsed time.

The value of SECNDS is accurate to the resolution of the system clock: Ø.Ø166... seconds for a 6ØHz clock and Ø.Ø2 seconds for a 5ØHz clock.

With 24 bits of precision for real values, this representation is accurate to the clock tick for values up to about two days in duration.

14.2.12 Radix-5Ø Conversions

14.2.12.1 IRAD5Ø

The IRAD5Ø subprogram performs conversions between Hollerith (text) strings and Radix-5Ø representation. (See Section 7-2.2.9 for details for the Radix-50 representation.)

IRAD5Ø may be called as a FUNCTION subprogram if the return value is desired, or as a SUBROUTINE subprogram if no return value is desired. The form of the call is

[1] This function is the preferred alternative to the 2- and 3-argument forms of the TIME subroutine call.

```
N = IRAD5Ø(ICNT,INPUT,OUTPUT)
```

or

```
CALL IRAD5Ø(ICNT,INPUT,OUTPUT)
```

where

ICNT   is the maximum number of characters to be converted (integer).

INPUT   is an ASCII (Hollerith) text string to be converted to Radix-5Ø.

OUTPUT   is the location for storing the results of the conversion.

N   is the number of characters actually converted.

Three characters of text are packed into each word of output. The number of output words modified is computed by the following expression (in integer mode).

```
(ICNT+2)/3
```

Thus, in a count if four is specified, two words of output will be written even if only a 1-character input string is given as an argument.

Scanning of input characters terminates on the first non-Radix-5Ø character encountered in the input string.

14.2.12.2 RAD5Ø

The RAD5Ø function subprogram provides a simplified way of converting ASCII data to Radix-5Ø representation. The form of the call is

```
X = RAD5Ø (string)
```

where

string   is the input ASCII string. Up to six characters are scanned and converted.

X   is a real variable to which the converted value is to be assigned.

The RAD5Ø function is equivalent to the following FORTRAN function.

```
        FUNCTION RAD5Ø(A)
        CALL IRAD5Ø (6,A,RAD5Ø)
        RETURN
        END
```

14.2.12.3 R5ØASC

The R5ØASC subprogram provides decoding of Radix-5Ø encoded values into ASCII strings. The form of the call is

```
        CALL R5ØASC(ICNT,IN,OUT)
```

where

ICNT   is the number of output characters to be produced.

IN   is the variable or array containing the encoded input. Note that (ICNT+2)/3 words will be read for conversion.

OUT   is the variable or array into which ICNT characters (bytes) will be placed.

If undefined Radix-5Ø code is detected, or the Radix-5Ø word exceeds the maximum value 174777_8, question marks will be placed in the output field.

14.2.13 SSWTCH

The sense switch (SSWTCH) subroutine can be used in FORTRAN programs to test the current status of specified bits of the console switch register. The subroutine is called by the statement

```
        CALL SSWTCH (I,J)
```

where:

I   is an integer value designating the console switch register bit to be tested; it must be within the range of Ø through 15. If it is outside of this range, no value is returned; no diagnostic is printed in this case.

J   is an integer variable that is to contain the value returned by the subroutine. It is set to 1 if bit I is 1 (the switch is up) or to 2 if bit I is Ø (the switch is down).

14.2.14 Character String Arguments to System Subroutines

String arguments to system subroutines must be stored in adjacent bytes. String values stored in integer arrays are not stored in adjacent bytes unless the /ON switch is used in compiling the program (see Section 7-9.2 and Chapter 7-12); such character strings are unacceptable as arguments to system subroutines.

The string values returned by the DATE and TIME subroutines are stored in adjacent bytes independent of the use of the /ON switch. Hence, the /ON switch is needed to produce correct results.

Object time format specifications (see Section 7-7.2∅) must also be stored in adjacent bytes.

String arguments to the ASSIGN subroutine must be stored in adjacent bytes and have a terminating zero byte. The FORTRAN system automatically appends a zero byte to character string arguments specified as Hollerith constants.

PART 7

CHAPTER 15

FORTRAN CALLING SEQUENCE CONVENTIONS

The calling sequence convention currently in use with FORTRAN on the PDP-11 family of computers is called "R5", and is used with all versions of the DOS/BATCH operating system.

The following points should be noted:

1. Version VØ6 (DOS/BATCH) of the FORTRAN Compiler implement subprogram calls $CALL, by means of an OTS threaded code routine, as described in Section 7-1Ø.2. For this reason, there is no compilation-time switch for specifying the convention to be used. FORTRAN routines compiled by the VØ6 Compiler may be linked with the R5 forms of the library.

2. FORTRAN routines compiled by earlier Compilers (V4A and before) may be linked with the R5 form of the library. It is recommended, however, that these routines be recompiled using the VØ6 Compiler.

15.1 PDP-11 FORTRAN "R5" CALLING SEQUENCE CONVENTION

The form of the subprogram call under the R5 convention is the same as that used with earlier versions of FORTRAN under the DOS/BATCH operating system.

15.1.1 The Call Site

The basic form of the call follows.

```
          JSR          %5,SUB            ;CALL SUBPROGRAM
          BR           NEXT              ;BRANCH OVER ARGUMENT LIST
          .WORD        ADR1              ;FIRST ARGUMENT ADDRESS
           .            .                  .
           .            .                  .
           .            .                  .
          .WORD        ADRN              ;NTH ARGUMENT ADDRESS
     NEXT:                               ;NEXT EXECUTABLE INSTRUCTION
```

Note that the low-order byte of the branch instruction contains the count of the number of arguments that follow. A maximum of 127 arguments are permitted.

15.1.2 Return

Control is returned from the called program unit to the calling program unit by restoring the stack pointer register (SP, register 6) to its value at the time of entry, if necessary, and executing the following.

```
          RTS          5%
```

15.1.3 Return Value Transmission

FUNCTION subprograms return values in general registers RØ through R3. The number of registers used is determined by the data type of the function's returned value, as follows.

Data Type	Value Returned in
BYTE (LOGICAL*1) LOGICAL INTEGER	RØ
REAL	RØ, R1
DOUBLE PRECISION COMPLEX	RØ, R1, R2, R3

Example:

The FORTRAN statement

```
          CALL SUB (A1,A2,A3)
```

results in a subprogram call equivalent to the following.

```
          .GLOBAL SUB
          JSR     %5,SUB
          BR      $Fnnnn
          .WORD   A1
          .WORD   A2
          .WORD   A3
  $Fnnnn:
```

Within the subprogram, arguments can be accessed by means of indexed references, using register 5 as the index register. In particular, the instruction

```
          MOVB     @%5, ...
```

obtains the number of arguments in the list,

```
          MOV      2%5, ...
```

obtains the address of the first argument, and, in general, the instruction

```
          MOV       2*n(%5), ...
```

obtains the address of the nth argument.

15.1.4 Null Arguments

Null arguments are represented in an argument list by using an address of -1 (177777$_8$). This address can be easily detected and generally assures that an error will result at execution time if a null argument is passed to a subprogram that is not prepared to accept it. Null arguments are included in the argument count as shown in the following examples:

FORTRAN Statement	Resulting Argument List
CALL SUB	BR .+2
CALL SUB ( )	BR .+4 .WORD -1
CALL SUB (A,)	BR .+6 .WORD A .WORD -1
CALL SUB (,B)	BR .+6 .WORD -1 .WORD B

15.2 MACROS FOR PDP-11 FORTRAN CALLING SEQUENCE

15.2.1 Introduction

Three macros have been defined to facilitate the writing of PDP-11 subprograms that interface to PDP-11 FORTRAN compiled programs. These provide three functions.

1) Calling a FORTRAN subprogram, including construction of the appropriate argument list.

2) Returning to a FORTRAN subprogram, including the transmission of the return value (if any).

3) Obtaining the value returned by a FORTRAN subprogram and moving it to the desired destination.

The goal of these macros is to make assembly language routines independent of the details of the FORTRAN calling sequence conventions.

The general form of each macro call is

```
        F4CALL Subroutine-name, argument-list, optional-label

        F4RTN Typecode,location

        F4VAL2 Typecode,location.
```

These macros are discussed in detail in the following sections.

In the following discussions reference will be made to the addressing mode of a macro argument. If unfamilar with the details of the PDP-11 addressing modes please refer to the appropriate processor handbook. Briefly, the following should be noted.

1) In some cases the code generated by these macros is dependent on the addressing mode of one or more arguments. The addressing mode is obtained by using the .NTYPE directive described in Part 6, MACRO.

2) Intuitively, the addressing mode is the six bit octal value that would be placed in either the source or destination field of a PDP-11 instruction.

3) The following cases are particularly important.

Argument Form	Addressing Mode
Simple identifier, e.g., X	67
Push to stack, e.g., -(%6)	46

15.2.2 Calling a FORTRAN subprogram

The form of the macro call is

```
        F4CALL      NAME,<ARGLIST>,LABEL
```

where

NAME is the name of the subprogram to be called.

ARGLIST is a list of addresses of the arguments.

LABEL is an optional identifier to be placed on the argument list. In most cases this label is not needed.

Examples:

```
        F4CALL      SUB,<A,B,C>            ;CALL SUBROUTINE SUB
                                            ;WITH ARGUMENTS A, B, AND C

        F4CALL      X                      ;CALL SUBROUTINE X WITH NO ARGUMENTS

        F4CALL      X,<A,>,L1
```

There are two parts to calling a subroutine: filling in the argument list (either at assembly time or at execution time) and transferring control to the subroutine. The latter is performed by

```
          JSR        REG,NAME
```

where the REGister used depends on the convention.

Filling in the argument list is slightly more complex. The argument list itself consists of a block of N+1 words where the low byte of the first word contains the number of arguments, N, and the next N words are the addresses of the arguments. There are four cases to be considered.

Case 1 - assembly time constants

In many cases the argument is at a fixed location and the address can be simply assembled into the argument list. This is done for all arguments with an addressing mode of 67.

Case 2 - argument value on stack

Often a result has been computed and left on a stack. In this case the address is the sum of the contents of the stack register (which need not be %6) and a constant offset. This is represented by an argument of the following form.

```
          Register,offset
```

For example:

```
          <%6,Ø>
          <%1,6>
```

For this situation the address of the data is computed by the sequence

```
     MOV              Register, Argument List Entry
     ADD              Offset, Argument List Entry
```

Note that the address is stored in the appropriate position in the argument list; if the offset is zero, no ADD instruction is needed or produced.

Case 3 - empty argument

In rare cases the user may want to fill in an argument address by code unrelated to the code that the F4CALL macro produces. Here it is necessary to designate the argument list label as part of the F4CALL. (Note that the first argument address goes at LABEL+2, the nth at LABEL+2*n, and so on.) Space for such an argument is reserved in the argument list and an address of -1 is assembled in.

Case 4 - general argument expression

If none of the above cases holds, then the argument address is obtained by generating the following instruction.

```
MOV             Argument,Argument List Entry
```

Note that any auto-incrementing or auto-decrementing indicated in a general argument expression will be performed as a result of the MOV instruction.

Putting these all together we illustrate with the example shown in Figure 7-14. Figure 7-15 illustrates how the F4CALL macro relates to the FORTRAN source code. Assume the following:

ARG1 — is the address of argument 1,
RØ — contains the address of argument 2,
R1 — contains a pointer to a two word table which contains the addresses of arguments 3 and 6,
R5 — points to the argument list of the call that entered this subroutine and the first argument of that call is to be the 4th argument of this call, and
- — the address of the fifth argument is the third word on the processor stack.
- — the seventh argument is filled in elsewhere.

Then write

```
F4CALL NAME, <ARG1,RØ,(R1)+,2(R5), <SP,6>, (R1)+,>,L
```

which would generate the following code for the PC calling sequence convention:

```
MOV             RØ,L+4
MOV             (R1)+,L+6
MOV             2(R5),L+1Ø
MOV             SP,L+12
ADD             #6,L+12
MOV             (R1)+,L+14
MOV             R5,-(SP)
MOV             #L,R5
JSR             PC,NAME
BR              $KPLST
```

Figure 7-14

Argument List Construction

```
              L:         .BYTE          7,Ø
                         .WORD          ARG1
                         .WORD          Ø
                         .WORD          Ø
                         .WORD          Ø
                         .WORD          Ø
                         .WORD          Ø
                         .WORD          -1
              $KPLST:    MOV            (SP)+,R5

     (Where $KPLST is a macro generated label.)
```

Figure 7-14 (Cont.)

Argument List Construction

```
             SUBROUTINE A(X,Y,Z)
             DIMENSION Z(1Ø)
             ...
             CALL B(I+J,X+C,D,Z,(I) ,X)
             ...
             END

might generate the following code using these macros:

     Compute I + J and place on stack (1 word)
     Compute X + C and place on stack (2 words)
     Compute address of Z(I) and leave in R0

     F4CALL B,<<SP,4>,<SP,Ø>,D,RØ,2(R5)>>
```

Figure 7-15

Relationship Between F4CALL and FORTRAN Source Code

15.2.3 Returning to a FORTRAN Program

The form of the macro call is

```
          F4RTN      Typecode,Location
```

where

Typecode is a single letter representing the data type as follows.

- B - Byte
- L - Logical
- I - Integer
- J - Double Integer
- R - Real
- D - Double Precision Real
- C - Complex

Location is either a simple variable name (addressing mode 67) or a stack pop addressing mode (2X).

The macro generates the move instruction needed to place the returned value in the correct position for recovery by the calling routine. (This is presently in the general registers %Ø-%3 but we wish to retain the option of using the floating registers on 11/45 FPP systems at a future date.) Finally the appropriate RTS instruction is generated.

If the macro is written with no arguments then only the RTS instruction is produced.

Note that the user code is responsible for assuring that the stack pointer (%6) is correctly positioned at runtime.

Several examples are shown in Figure 7-16.

```
                          Example 1

Macro call:                 F4RTN

Expanded code-              RTS         R5


                          Example 2

Macro call:                 F4RTN       B,(%1)+

Expanded code-              MOVB        (%1)+,RØ
                            RTS         R5



                          Example 3

Macro call:                 F4RTN       R,LOC+4

Expanded Code-              MOV         LOC+4,RØ
                            MOV         LOC+4+2,R1
                            RTS         R5
```

Figure 7-16

Returning to the FORTRAN Program

15.2.4 Obtaining the Returned Value

The form of the macro call is

```
        F4VAL2     Typecode,Location
```

where

Typecode is a single letter representing the data type as in the F4RTN macro, and location is either a simple variable name (mode 67) or a stack push addressing mode (4X).

The expansion produced moves the returned value into the designated location or onto the designated stack.

Examples are shown in Figure 7-17.

Note that code generated for this macro is not currently call sequence dependent. This macro is included to isolate the entire call sequence mechanism so that, for example, floating-point values may some day be returned in FPP registers.

```
                              Example 1
Macro call:                             F4VAL2      I,-(SP)

Generated code-                         MOV         RØ,-(S)


                              Example 2
Macro call:                             F4VAL2      C,VAR

Generated code-                         MOV         RØ,VAR
                                        MOV         R1,VAR+2
                                        MOV         R2,VAR+4
                                        MOV         R3,VAR+6
```

Figure 7-17
Return Value Transmission

15.2.5 Obtaining and Using the Macros

The macros are distributed as part of the system macro file SYSMAC.SML. They are defined in the user module in the following way.

The macro .F4DEF must be obtained from the system macro file via a

```
          .MCALL     .F4DEF
```

directive. Then the macro .F4DEF must be invoked with an argument specifying the calling form to be defined: Ø for the R5 form. Its action is simply to define the macros as appropriate for the value of its argument.

Thus to obtain the R5 form call the user includes the following.

```
          .MCALL      .F4DEF
          .F4DEF      Ø
```

As a side effect .F4DEF also defines the variable .F4SEQ to have the value of its argument. This symbol may then be used to conditionally assemble sections of code which, for reasons not anticipated here, may be call sequence dependent.

Finally the .F4DEF macro may be called without an argument in which case the value of .F4SEQ is assumed to be already defined as Ø, and its value is used to define the remaining macros. This is useful, for example, where the value of .F4SEQ is defined in a separate parameterization file used with a collection of routines.

15.2.6 Programming Cautions and Notes

In addition to the macros F4CALL, F4RTN and F4VAL2, several additional macros are defined for use by the user level macros and several variables are used for communication between macros at assembly time. All of these internal macro and variable names begin with the period character. Users are urged to not use identifiers beginning with the period character.

It is a general policy that use of identifiers containing the period character is to be specified exclusively by DEC. Users are urged to avoid all such names in order to assure that name conflicts will not be introduced in future releases of existing or new software.

In the R5 form of calling convention FORTRAN register 5 always points to the argument list of the routine calling the current one and this pointer is conveniently saved and restored by the JSR R5/RTS R5 pair. Fortunately the new FORTRAN generated code will not require that the argument pointer be also available in register 5 and so does not need to explicitly save/restore R5.

For those concerned about minimizing core and willing to expend more care in checking the compatibility, it may be worthwhile to try the following call.

```
          .F4DEF 2
```

This is identical to .F4DEF 1 except the explicit save/restore of R5 is not generated.

PART 7

CHAPTER 16

FORTRAN TRACE PACKAGE

16.1 TRACE PACKAGE[1]

A FORTRAN trace package is provided as a library-called TRCLIB.OBJ that provides a FORTRAN-level trace of program execution. Actions that can be traced include assignment to variables, call and return from subprograms, and transfers of control within a program unit.

The library consists of two modules: TRACEX, which is a special version of selected OTS threaded code service routines, and TRACEF, which is a DEC-supplied FORTRAN subroutine for performing formatted tracing of program action. If desired, a user may supply his own version of the SUBROUTINE TRACEF for altering the trace dump to his own needs.

16.2 TRACE OUTPUT DESCRIPTION

A summary of the types of trace provided and the trace output generated is shown in Table 7-20.

16.3 SELECTIVE CONTROL OF PROGRAM TRACING

Trace output for each type of action may be independently enabled or disabled either under control of the PDP-11 console switches or under program control.

The normal or default mode is to control tracing using the console switches. In this mode each time the trace package is entered the console switches are interrogated to determine whether a trace output is to be produced. Each type of trace is output only if the corresponding console switch is in the up position. For example, subprogram entry information is printed only if console switch 1 is up, subprogram return only if switch 2 is up, and so on.

Tracing may be placed under program control by means of the FORTRAN subroutine TRCTRL. This subprogram is called as follows.

```
CALL TRCTRL (ITYPE,ICODE)
```

[1] This package depends on the implementation of FORTRAN compiled output using threaded code techniques such as are found in VØ6. It may not be compatible with later versions.

Table 7-20

Trace Output Description

Trace Type	Description	Trace Output
1	Entry to FORTRAN subprogram	ENTER name
2	Return to calling program	RETURN TO name
3	Sequence number of statement about to be executed	SEQ. nnnn
4	Transfer of control resulting from any type of GOTO statement	GOTO nnnn
5	Arithmetic assignment	type = value
6	Arithmetic assignment to variables specified in CALL TRCLST statement	type = value
7	ASSIGN statement	ASSIGN = nnnn

where

name   is a subroutine or function name

nnnn   is a statement sequence number

type   is the type of value being assigned: INTEGER,REAL, etc. for arithmetic assignment

value   is the value assigned, printed in the appropriate format

where

ITYPE   is an integer value designating the type of trace output to be effected (see Table 7-20).

ICODE   is an integer value designating whether tracing is to be enabled or disabled (1=enable, 0=disable).

Special cases of the above:

CALL TRCTRL (Ø,Ø)   places tracing under program control with all tracing disabled.

CALL TRCTRL (Ø,1)   places tracing under program control with all tracing enabled.

CALL TRCTRL (-1,Ø)   places tracing under control of the console switches.

Note that more than one type of trace may be enabled by multiple calls to TRCTRL. For example, to trace only subprogram calls and returns, the two statements

```
CALL TRCTRL (1,1)
CALL TRCTRL (2,1)
```

may be executed. An initialization call of CALL TRCTRL (Ø,Ø) is unnecessary because any call to TRCTRL that changes the tracing mode from switch control to program control automatically disables all tracing except that specified by the call.

In many cases it is desirable to trace assignment statements for a designated set of variables only. This can be accomplished by the TRCLST entry. The form of this call is

CALL TRCLST (V_1,V_2,...,V_n)

where each V_i is the name of a variable to be traced. (Note that only single elements can be specified. An unsubscripted array name causes only the first element of the array to be traced, not the entire array.) Up to 2Ø variable names may be specified for tracing in the program (regardless of the number of TRCLST calls); variables entered in excess of 2Ø are simply ignored. Variables may be deleted from the trace list by executing the following statement.

CALL TRCDEL (V_1,V_2,...,V_n)

An attempt to delete from the trace list a variable that does not exist in the list causes no problems.

16.4 TRACE OUTPUT DEPENDENCE ON COMPILATION OPTIONS

If a FORTRAN program or subprogram is compiled with the /SU option in effect, sequence number information is not available in the compiled output. In this case all trace output requiring a sequence number prints a Ø for that number.

If a FORTRAN program or subprogram is compiled with the /OP option set to a value other than Ø, many assignment operations will be undetectable by the trace package. To ensure that all assignment actions are traceable, use /OP:Ø in the compiler control string.

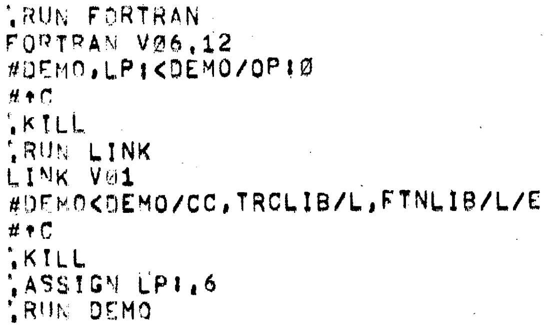
16.5 USAGE

No modifications to FORTRAN source programs are required to obtain tracing under switch control. If control of tracing through TRCTRL, TRCLST, or TRCDEL is desired, however, appropriate calls to those entry points must be inserted in the source program where desired.

To provide the tracing features described above, the library TRCLIB is included in the input file portion of the LINK command string following all other user's files, but before the normal library, FTNLIB.

Example:

A complete example of the commands needed to trace a demonstration program, DEMO.FTN, and the output provided by the TRACEF subroutine, is presented in Figures 7-18, 7-19 and 7-20.



```
.RUN FORTRAN
FORTRAN VØ6.12
#DEMO,LP:<DEMO/OP:Ø
#↑C
.KILL
.RUN LINK
LINK VØ1
#DEMO<DEMO/CC,TRCLIB/L,FTNLIB/L/E
#↑C
.KILL
.ASSIGN LP:,6
.RUN DEMO
```

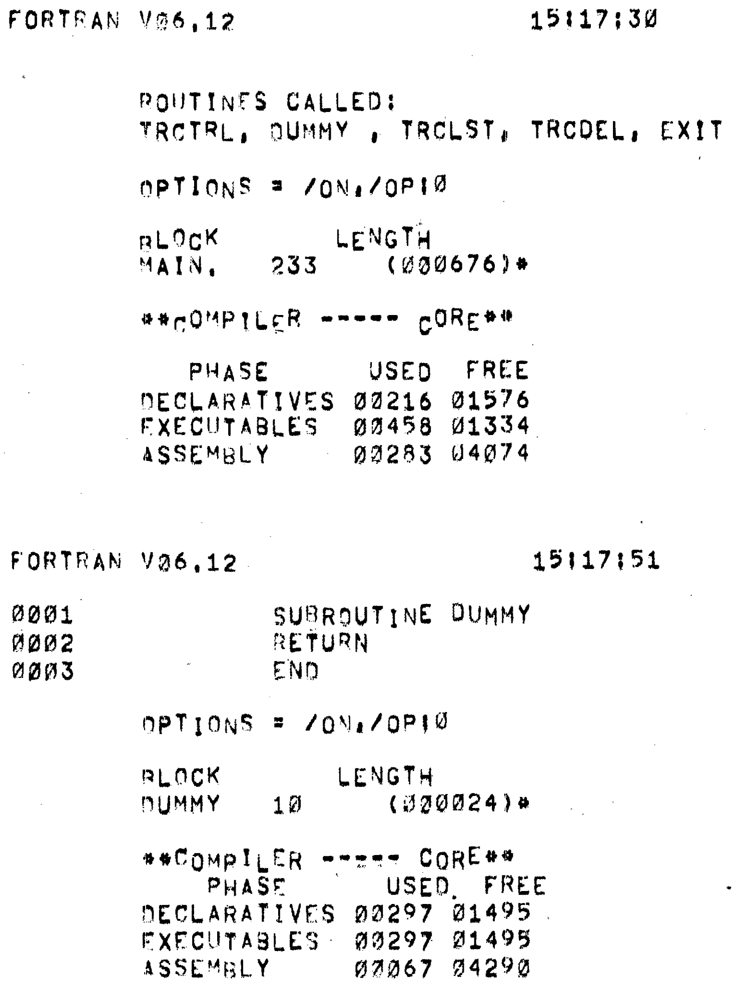
Figure 7-18

Console Commands to trace DEMO.FTN on the Line Printer

```
ØØØ1              BYTE B
ØØØ2              LOGICAL L
ØØØ3              INTEGER I
ØØØ4              REAL R
ØØØ5              DOUBLE PRECISION D
ØØØ6              COMPLEX C
          C
          C    TRACE EVERYTHING
          C
ØØØ7              CALL TRCTRL(Ø,1)
          C
          C    TYPES 1, 2, AND 3; CALL, RETURN AND SEQUENCE
          C
ØØØ8              CALL DUMMY
          C
          C    TURN OFF SEQUENCE TRACE
          C
ØØØ9              CALL TRCTRL(3,Ø)
ØØ1Ø              CALL DUMMY
          C
          C    TYPE 4; GOTO TRACE
          C
ØØ11              GOTO 1Ø
ØØ12      1Ø      CONTINUE
          C
          C    TYPE 5; ARITHMETIC ASSIGNMENT TRACE
          C
ØØ13              B = 1
ØØ14              L = .TRUE.
ØØ15              I = 2
ØØ16              R = 1.5
ØØ17              D = 2.3DØ
ØØ18              C = (1.1,-2.3)
          C
          C    TYPE 6; SELECTED VARIABLE TRACE
          C
ØØ19              CALL TRCTRL(5,Ø)
ØØ2Ø              Y = 1.
ØØ21              CALL TRCLST(Y,B)
ØØ22              Y = 2
ØØ23              CALL TRCDEL(Y,I)
ØØ24              Y = 3.
          C
          C    TYPE 7; ASSIGN STATEMENT TRACE
          C
ØØ25              ASSIGN 4Ø TO K
          C
          C    MORE TYPE 4; GOTO TRACE
          C
ØØ26              GOTO K
ØØ27      4Ø      M = 2
ØØ28      5Ø      GOTO (5Ø,6Ø) M
ØØ29      6Ø      CONTINUE




ØØ3Ø              CALL EXIT
ØØ31              END
```

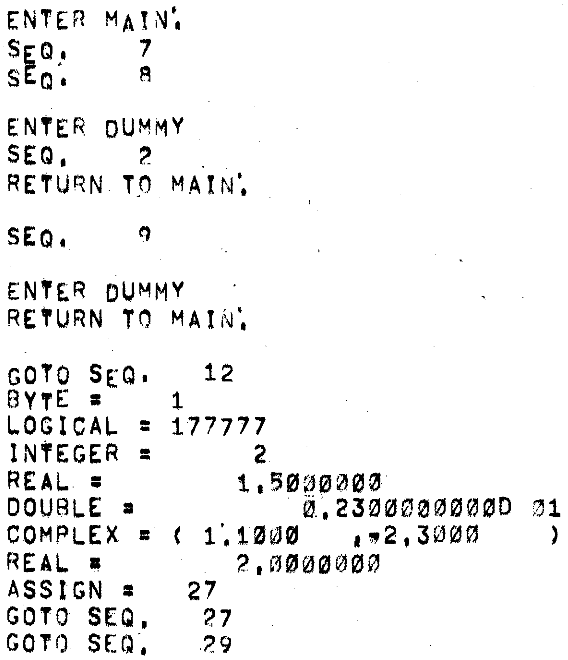
Figure 7-19

Compilation Listing of DEMO.FIN



```
FORTRAN VØ6.12                          15:17:3Ø

          ROUTINES CALLED:
          TRCTRL, DUMMY , TRCLST, TRCDEL, EXIT

          OPTIONS = /ON,/OP:Ø

          BLOCK        LENGTH
          MAIN.    233    (ØØØ676)*

          **COMPILER ----- CORE**

             PHASE        USED  FREE
          DECLARATIVES ØØ216 Ø1576
          EXECUTABLES  ØØ458 Ø1334
          ASSEMBLY     ØØ283 Ø4074


FORTRAN VØ6.12                          15:17:51

ØØØ1              SUBROUTINE DUMMY
ØØØ2              RETURN
ØØØ3              END

          OPTIONS = /ON,/OP:Ø

          BLOCK        LENGTH
          DUMMY    1Ø     (ØØØØ24)*

          **COMPILER ----- CORE**
              PHASE       USED  FREE
          DECLARATIVES ØØ297 Ø1495
          EXECUTABLES  ØØ297 Ø1495
          ASSEMBLY     ØØØ67 Ø429Ø
```

Figure 7-19 (Cont.)



```
ENTER MAIN.
SEQ.    7
SEQ.    8

ENTER DUMMY
SEQ.    2
RETURN TO MAIN.

SEQ.    9

ENTER DUMMY
RETURN TO MAIN.

GOTO SEQ.   12
BYTE =    1
LOGICAL = 177777
INTEGER =      2
REAL =        1.5ØØØØØØ
DOUBLE =          Ø.23ØØØØØØØØD Ø1
COMPLEX = ( 1.1ØØØ   ,-2.3ØØØ     )
REAL =        2.ØØØØØØØ
ASSIGN =   27
GOTO SEQ.   27
GOTO SEQ.   29
```

Figure 7-20

Trace Output of Program DEMO

PART 7

CHAPTER 17

FORTRAN DEVICE TABLE LISTING

```
        .TITLE  $DV8Ø9
        GLOBL   $DEVTB
        .CSECT
;
;       $DEVTB  VØØ9A
;
; COPYRIGHT 1971,1972 DIGITAL EQUIPMENT CORPORATION, MAYNARD,MASS
;
;THESE ARE THE FORTRAN DEVICE TABLE ENTRIES
;WITH THE DEVICE TABLE HEADER AND ENTRY VECTOR
;
        .IFNDF  RSX
        WORD    DEVM3                   ;ADDR OF ENTRY FOR ERR LOG DEVICE
                                        ;ERROR LOG DEVICE IS -3(DEVM3)
        .WORD   DEVERR                  ;ADDR OF ENTRY FOR ERR MSG FILE
        .ENDC
$DEVTB: .WORD   8.                      ;NUMBER OF ENTRIES IN ENTRY VECTOR
        .IFNDF  RSX
        .WORD   -3                      ;DEVICE NUM OF ERROR LOGGING DEVICE
        .ENDC
        .IFDF   RSX
        .WORD   5.                      ;DEVICE NUM OF ERROR LOGGING DEVICE
        .ENDC
;
;THE DEVICE TAPLE ENTRY VECTOR
;
        .WORD   DEV1                    ;ADDR OF DEVICE 1 ENTRY
        .WORD   DEV2                    ;ADDR OF DEVICE 2 ENTRY
        .WORD   DEV3                    ;ADDR OF DEVICE 3 ENTRY
        .WORD   DEV4                    ;ADDR OF DEVICE 4 ENTRY
        .WORD   DEV5                    ;ADDR OF DEVICE 5 ENTRY
        .WORD   DEV6                    ;ADDR OF DEVICE 6 ENTRY
        .WORD   DEV7                    ;ADDR OF DEVICE 7 ENTRY
        .WORD   DEV8                    ;ADDR OF DEVICE 8 ENTRY
;
;
;
;ENTRY 1 OF DEVICE TABLE
;
DEV1:   .WORD   Ø                       ;LINK BLOCK PTR
        .IFNDF  RSX
        .RAD5Ø  /SY /                   ;PHYSICAL DEVICE NAME DEFAULT
        .ENDC
        .IFDF   RSX
        .RAD5Ø  /KB/
        .ENDC
        .BYTE   Ø                       ;HOW OPEN SWITCH


        .BYTE   Ø                       ;UNIT NUM DEFAULT
        .RAD5Ø  /FOR/                   ;DEFAULT FILE NAME
        .RAD5Ø  /ØØ1/
        .RAD5Ø  /DAT/                   ;DEFAULT EXTENSION
        .BYTE   233             ;NO AUTO DEL, GROUP & OTHER READ/RUN ONLY
        .BYTE   Ø               ;DEVICE STATUS SWITCH
        .BYTE   Ø               ;MODE OF I/O = FUNCN WORD (RANDOM)
        .BYTE   Ø               ;STATUS OF I/O
        .WORD   Ø               ;RECORD COUNT = BLOCK NUM (RANDOM)
        .WORD   Ø               ;BUFF ADDR (RANDOM)
        .WORD   Ø               ;BUF LEN (RANDOM)
        .WORD   Ø               ;ASSOCIATED VAR ADDR (FROM DEFINE FILE)
        .WORD   Ø               ;NUM RECORDS IN FILE (FROM DEFINE FILE)
        .WORD   Ø               ;RECORD LENGTH (FROM DEFINE FILE)
        .WORD   Ø               ;USER ID CODE
        .WORD   Ø               ;ERROR VAR ADDR (FROM SETFIL)
;
;
;
;ENTRY 2 OF DEVICE TABLE
;
DEV2:   .WORD   Ø
        .IFNDF  RSX
        .RAD5Ø  /SY /
        .ENDC
        .IFDF   RSX
        .RAD5Ø  /TT/
        .ENDC
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ2/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;
;ENTRY 3 OF DEVICE TABLE
;
DEV3:   .WORD   Ø
        .IFNDF  RSX
        .RAD5Ø  /SY /
        .ENDC
        .IFDF   RSX
        .RAD5Ø  /PP/
        .ENDC
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ3/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;
;ENTRY 4 OF DEVICE TABLE
;
DEV4:   .WORD   Ø
        .RAD5Ø  /PR/
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ4/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;ENTRY 5 OF DEVICE TABLE
;
;
DEV5:   .WORD   Ø
        .RAD5Ø  /LP /
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ5/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;ENTRY 5 OF DEVICE TABLE
;
;
DEV5:   .WORD   Ø
        .RAD5Ø  /KB /
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ6/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;ENTRY 7 OF DEVICE TABLE
;
;
DEV7:   .WORD   Ø
        .IFNDF  RSX
        .RAD5Ø  /SY /
        .ENDC
        .IFDF   RSX
        .RAD5Ø  /UD/
        .ENDC
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ7/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;
;ENTRY 8 OF DEVICE TABLE
;
;
DEV8:   .WORD   Ø
        .IFNDF  RSX
        .RAD5Ø  /BI /
        .ENDC
        .IFDF   RSX
        .RAD5Ø  /AD/
        .ENDC
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/
        .RAD5Ø  /ØØ8/
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
        .IFNDF  RSX
;
;ENTRY = 3 OF DEVICE TABLE(ERROR LOGGING DEVICE)
;SPECIAL ENTRY USED AS DOS-11 ERROR LOGGING DEVICE
;LOGICAL UNIT NUMBER = -3,  LOGICAL DEVICE NAME = CMO
;
DEVM3:  .WORD   Ø                       ;LINK BLOCK PTR
        .RAD5Ø  /KB /                   ;DEFAULT PHYSICAL DEVICE NAME
        .BYTE   Ø,Ø
        .RAD5Ø  /FOR/                   ;DEFAULT FILE NAME
        .RAD5Ø  /CMO/                   ;FORCMO.DAT
        .RAD5Ø  /DAT/
        .BYTE   233,Ø,Ø,Ø
        .WORD   Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
;
;
;
;SPECIAL ENTRY FOR ERROR PROCESSORS MSG FILE
;
        .WORD   Ø                       ;LINK BLOCK ERR RTN ADDR
DEVERR: .WORD   Ø                       ;LINK PTR
        .RAD5Ø  /ERR/                   ;LOG DATA SET NAME
        .BYTE   1                       ;PHYSICAL DS NAME FOLLOWS
        .BYTE   Ø                       ;UNIT NUM
        .RAD5Ø  /SY /                   ;PHYSICAL DEVICE NAME
                                        ;DEFAULT TO SYSTEM DEVICE
        .WORD   Ø                       ;FILE BLOCK ERROR RETURN ADDR
        .BYTE   4                       ;HOW TO OPEN (OPENI)
        .BYTE   Ø                       ;ERROR RTN CODE
        .RAD5Ø  /FOR/                   ;FILE NAME
        .RAD5Ø  /RUN/
        .RAD5Ø  /DGN/
        .BYTE   1                       ;USER ID CODE
        .BYTE   1
        .BYTE   322,Ø                   ;ALLOW ONLY INPUT ACCESS
        .WORD   4                       ;FUNCTION WORD (READ)
        .WORD   Ø                       ;BLOCK NUM
        .WORD   Ø                       ;BLOCK ADDR
        .WORD   Ø                       ;BLOCK LENGTH
;
;
        .ENDC
        .END
```