FORTRAN IV XVM LANGUAGE MANUAL

DEC-XV-LF4MA-A-D

XVM
Systems

digital

FORTRAN IV XVM LANGUAGE MANUAL

DEC-XV-LF4MA-A-D

digital equipment corporation · maynard. massachusetts

First Printing, December 1975

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

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

The following are trademarks of Digital Equipment Corporation:

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

CONTENTS

CONTENTS (Cont.)

Page

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

TABLES

LIST OF ALL XVM MANUALS

The following is a list of all XVM manuals and their DEC numbers, including the latest version available. Within this manual, other XVM manuals are referenced by title only. Refer to this list for the DEC numbers of these referenced manuals.

BOSS XVM USER'S MANUAL	DEC-XV-OBUAA-A-D
CHAIN XVM/EXECUTE XVM UTILITY MANUAL	DEC-XV-UCHNA-A-D
DDT XVM UTILITY MANUAL	DEC-XV-UDDTA-A-D
EDIT/EDITVP/EDITVT XVM UTILITY MANUAL	DEC-XV-UETUA-A-D
8TRAN XVM UTILITY MANUAL	DEC-XV-UTRNA-A-D
FOCAL XVM LANGUAGE MANUAL	DEC-XV-LFLGA-A-D
FORTRAN IV XVM LANGUAGE MANUAL	DEC-XV-LF4MA-A-D
FORTRAN IV XVM OPERATING ENVIRONMENT MANUAL	DEC-XV-LF4EA-A-D
LINKING LOADER XVM UTILITY MANUAL	DEC-XV-ULLUA-A-D
MAC11 XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMLAA-A-D
MACRO XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMALA-A-D
MTDUMP XVM UTILITY MANUAL	DEC-XV-UMTUA-A-D
PATCH XVM UTILITY MANUAL	DEC-XV-UPUMA-A-D
PIP XVM UTILITY MANUAL	DEC-XV-UPPUA-A-D
SGEN XVM UTILITY MANUAL	DEC-XV-USUTA-A-D
SRCCOM XVM UTILITY MANUAL	DEC-XV-USRCA-A-D
UPDATE XVM UTILITY MANUAL	DEC-XV-UUPDA-A-D
VP15A XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVPAA-A-D
VT15 XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVTAA-A-D
XVM/DOS KEYBOARD COMMAND GUIDE	DEC-XV-ODKBA-A-D
XVM/DOS READER'S GUIDE AND MASTER INDEX	DEC-XV-ODGIA-A-D
XVM/DOS SYSTEM MANUAL	DEC-XV-ODSAA-A-D
XVM/DOS USERS MANUAL	DEC-XV-ODMAA-A-D
XVM/DOS V1A SYSTEM INSTALLATION GUIDE	DEC-XV-ODSIA-A-D
XVM/RSX SYSTEM MANUAL	DEC-XV-IRSMA-A-D
XVM UNICHANNEL SOFTWARE MANUAL	DEC-XV-XUSMA-A-D

PREFACE

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. It is also widely used in a variety of applications including process control, information retrieval, and commercial data processing.

This document describes the form of the basic elements of the FORTRAN program, the FORTRAN statements. The document is a reference manual, and although it may well be used by an inexperienced FORTRAN programmer, it is not designed to function as a tutorial manual.

This document serves as the FORTRAN Language Reference Manual for the XVM/DOS and XVM/RSX Operating Systems. There is very little difference in the language of programs written for XVM/DOS and XVM/RSX. The differences that do exist are in the input/output statements:

1. XVM/DOS permits a Magtape-like file to be created on disk and REWIND and BACKSPACE statements to this file; XVM/RSX does not.
2. The CALL DEFINE statement, which creates and opens direct access files on disk, functions differently in the two systems.
3. The statements used to open, close, rename, and delete named files are formed differently under XVM/DOS and XVM/RSX.
4. XVM/DOS FORTRAN supplies implied or default file names for sequential access files; XVM/RSX FORTRAN does not.

A companion manual to this document is the FORTRAN IV XVM Operating Environment Manual. Also, referenced within this manual is the XVM/RSX System Manual.

DOCUMENTATION CONVENTIONS

Throughout this manual the following notations are used to denote special non-printing characters:

⇥	Tab character (TAB key or CTRL/I key combination)
Δ (delta)	Space character (SPACE bar)

SYNTAX NOTATION

The following conventions are used in the description of FORTRAN statement syntax.

1. Upper case words and letters, as well as punctuation marks other than those described in this section, are written as shown.

2. Lower case words indicate that a value is to be substituted. The accompanying text specifies the nature of the item to be substituted, e.g., integer variable or statement label.

3. Square brackets ([]) enclose optional items.

4. An ellipsis (...) indicates that the preceding item or bracketed group may be repeated any number of times.

For example, if the description were

```
CALL sub [ (a[,a]...) ]
```

then all of the following would be correct:

```
CALL TIMER
CALL INSPCT (I,J,3.0)
CALL REGRES (A)
```

CHAPTER 1

INTRODUCTION TO XVM FORTRAN

1.1 LANGUAGE OVERVIEW

The FORTRAN (FORmula TRANslation) language is exceptionally useful in scientific and mathematical applications. It provides the user with a means of solving equations and formulas rapidly and easily, and of performing large numbers of mathematical calculations. XVM FORTRAN conforms to the specifications for American National Standard FORTRAN X3.9-1966 with a number of additions and some exceptions. (See Appendix E, Extensions and Restrictions to ANSI 1966 Standard.)

1.2 ELEMENTS OF A FORTRAN PROGRAM

A FORTRAN program consists of FORTRAN statements and optional comments. The statements are arranged into logical units called program units (either a main program or a subprogram). Subprograms are external to the main program and cannot be embedded within it. One or more program units (one main program and possibly one or more subprograms) comprise the executable program. The FORTRAN Compiler translates source programs (written in the form described in this manual) into object programs (relocatable binary programs) which can then be loaded and executed by a computer.

1.2.1 Statements

Statements are grouped into two general classes: executable and non-executable. Executable statements describe the action of the program; nonexecutable statements provide the FORTRAN Compiler with instructions required in the translating process, such as, the size of an array, whether the program is a subroutine or not, and describe data arrangement and characteristics.

There are six categories of FORTRAN statements and a chapter in this manual is devoted to each:

Statement Type	Chapter	General Function
Assignment Statements	3	Assign values to symbols.
Control Statements	4	Govern the sequence in which executable statements are performed.
Specification Statements	5	Define characteristics of variables and arrays necessary for initialization.
Input/Output Statements	6	Govern the transfer of information between the computer and peripheral (I/O) devices.
FORMAT Statements	7	Describe the format in which data fields are transmitted or received.
Subprogram Statements	8	Define classes of subprograms and entry points to them.

Statements are divided into physical sections called lines. A line is a string of up to 72 characters. If a statement is too long to be contained on one line, it may be continued on one or more additional lines, called continuation lines. A continuation line is identified by the presence of a continuation character in the sixth column of that line. (For further information concerning continuation characters, see Section 1.3.4, Continuation Field.)

Certain statements may be identified by statement labels so that other statements can refer to them, either for the information they contain or to transfer control to them. A statement label has the form of an integer number placed in the first five columns of a statement's initial line. Blank lines are illegal.

1.2.2 Comments

Comments do not affect the meaning of the program in any way, but are a documentation aid to the programmer. They should be used freely to describe the actions of the program, to identify program sections and processes, and to provide greater ease in reading the source program listing. The letter C in the first column of a source line identifies that line as a comment.

1.2.3 The FORTRAN Character Set

The FORTRAN character set consists of:

1. The letters A through Z
2. The numerals 0 through 9
3. The following special characters:

Character	Name
Δ	Space or blank
⇥	Tab
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe (Single Quote)
"	Double Quote
$	Dollar Sign
[	Left (Open) Bracket
]	Right (Close) Bracket
:	Colon
;	Semicolon
#	Sharp Sign (Pound Sign)
@	At Sign

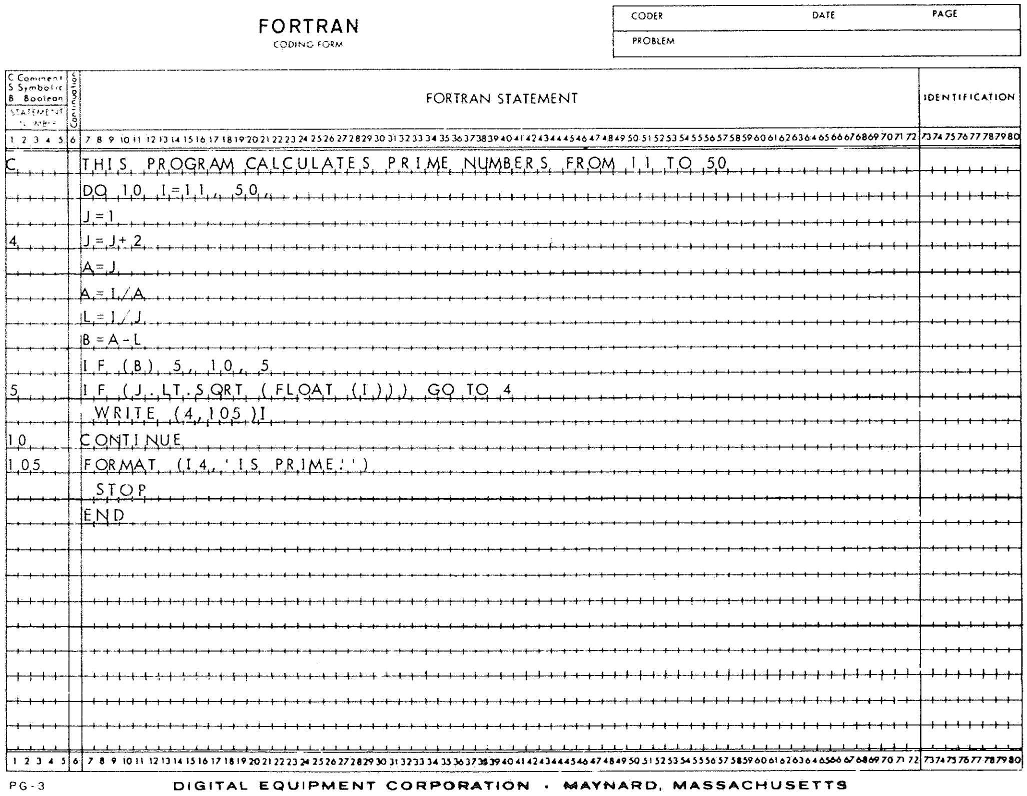
Other printable characters may appear in a FORTRAN statement only as part of a Hollerith constant, alphanumeric literal, or in a comment.

1.3 FORMATTING A FORTRAN LINE

The formatting of a FORTRAN line is nearly the same for programs written on FORTRAN coding forms and punched into cards or paper tape for presentation to the compiler and those entered from a terminal using a text editor. Only the method of formatting differs.

1.3.1 Using FORTRAN Coding Forms and Punched Cards

A FORTRAN line is divided into fields for statement labels, continuation indicators, statement text and sequence numbers. Each column represents a single character. The usage of each type of field is described in subsequent sections.



FORTRAN
CODING FORM

CODER DATE PAGE
PROBLEM

C Comment S Symbolic B Boolean | STATEMENT NUMBER | Continuation | FORTRAN STATEMENT | IDENTIFICATION

```
C     THIS PROGRAM CALCULATES PRIME NUMBERS FROM 11 TO 50
      DO 10 I=11, 50
      J=1
4     J=J+2
      A=J
      A=I/A
      L=I/J
      B=A-L
      IF (B) 5, 10, 5
5     IF (J.LT.SQRT (FLOAT (I))) GO TO 4
       WRITE (4,105)I
10    CONTINUE
105   FORMAT (I4,'IS PRIME:')
       STOP
      END
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

PG-3 DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS

Figure 1-1
FORTRAN Coding Form

1.3.2 Using a Text Editor and an On-Line Terminal

When creating a source program via a terminal, using a text editor, the user may type the lines on a "character-per-column" basis, as just described, or the user may use the TAB character to facilitate formatting the lines.

If a TAB character appears at the beginning of a line, possibly preceded by a statement label, and the following character is a non-zero digit, the Compiler treats the digit as a continuation indicator. If the following character is not a digit, the Compiler treats it as the first character of the statement text. If the line is a continuation line, the statement text begins with the character following the continuation character. If the continuation character is a "0", the line is an initial line.

While many text editors and terminals advance the terminal print carriage to a predefined print position when the TAB character is typed, this action is not related to the interpretation of the TAB character described above. Therefore, the TAB positions the next character at logical column 6 or 7, if the character is or is not a continuation character.

Formatting of the following lines can be accomplished in either of the following ways:

```
   ⇥1ΔHOLD,MOVE,DECODE          or   ΔΔΔΔΔ1ΔHOLD,MOVE,DECODE
 C ⇥INITIALIZEΔARRAYS           or   CΔΔΔΔΔINITIALIZEΔARRAYS
 10⇥W=3                         or   10ΔΔΔΔW=3
   ⇥SEL(1)=1.11200022D0         or   ΔΔΔΔΔΔSEL(1)=1.11200022D0
```

where:

⇥ represents a TAB character (CTRL/I), and
Δ represents a space character (SPACE bar).

The space character may be used in a FORTRAN statement to improve legibility of the line; the FORTRAN Compiler ignores all spaces in a statement field except those within a Hollerith constant or a Hollerith field in a format specification (for example, GOΔTO and GOTO are equivalent). The TAB character in a statement field is legal only if it appears prior to logical column 7; in the source listing produced by the Compiler, the TAB causes the following character to be printed in column 9.

1.3.3 Statement Number Field

A statement number, or statement label, consists of one to five decimal digits placed in the first five columns of a statement's initial line. Spaces and leading zeros are ignored. An all-zero statement label is prohibited. Statement labels need not appear in any numerical order within the program, and thus have no effect on the sequence of execution.

Any statement to which reference is made by another statement must have a label. No two statements within a program unit can have the same label.

1.3.3.1 Comment Indicator - The letter C may be placed in column 1 of this field to indicate that the line is a comment. The Compiler prints the contents of that line in the source program listing and then ignores the line.

1.3.4 Continuation Field

When spaces are used at the beginning of a line, column 6 of a FORTRAN line is reserved for a continuation indicator, i.e., any character except zero or space in this column is recognized as a continuation indicator. A common practice, though not required, is to place a zero in column 6 of a statement's initial line (to indicate that continuation lines follow) and then to number the continuation lines sequentially, placing the numbers in column 6 as continuation indicators.

When a TAB character appears at the beginning of a line, a digit from 1 - 9 must follow the TAB to indicate a continuation line. A statement may be divided into distinct lines at any point. The characters beginning in column 7 of a continuation line are considered to follow the last character of the previous line as if there were no break at that point. There are three exceptions. First, the DO statement must appear entirely on one line. Second, the assignment statement and the Arithmetic IF statement may be broken into continuation lines only at specific points. The continuation rules for these two statements are described in Sections 3.1 and 4.2.1.

Comment lines cannot be continued. All comment lines must begin with the letter C in column 1.

1.3.5 Statement Field

The text of a FORTRAN statement is placed in columns 7 through 72. Because the Compiler ignores spaces (except in Hollerith constants and Hollerith fields of format specifications, the user may space the text in any way desired for maximum legibility.

1.3.6 Sequence Number Field

A sequence number or other identifying information may appear in columns 73-80 of any line in a FORTRAN program. The characters in this field are ignored by the compiler.

CAUTION

Text may be ignored with no warning message if a line accidentally extends beyond character position 72.

1.4 PROGRAM UNIT STRUCTURE

Figure 1-2 provides a graphic representation of the rules for statement ordering. In this figure vertical lines separate statement types which may be interspersed, such as DATA and executable statements; horizontal lines indicate statement types that may not be interspersed, such as DATA and IMPLICIT statements. In general, specification statements and subprogram declarators must precede executable statements.

Comment[1] Lines	FUNCTION, SUBROUTINE or BLOCK DATA Statement
	IMPLICIT Statements
	INTEGER, REAL, LOGICAL, DOUBLE PRECISION and DOUBLE INTEGER Statements in any order
	DIMENSION Statements
	COMMON Statements
	EQUIVALENCE and EXTERNAL Statements in any order
	DATA Statements
	Statement Function Definitions
	All Other Statements
END Statement	

[1] Comment lines must not intervene between a statement's initial line and its continuation line(s), or between successive continuation lines.

Figure 1-2
Required Order of Statements and Lines

The maximum size of a program unit which contains executable code is constrained by physical addressing limitations of the XVM hardware to be 8176 storage words or less. This includes all arrays which do not appear in COMMON blocks. COMMON blocks, and arrays which appear within them, may be as large as 32,767 words.

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

The basic components of FORTRAN expressions are:

1. Constants
2. Variables
3. Arrays
4. Expressions
5. Function References

A brief definition of each of these basic components follows.

1. A constant is a data value that is self-defining and that cannot change.
2. A variable is a symbolic name that represents a stored value.
3. An array is a group of data, stored contiguously, that can be referred to individually or collectively. Individual values are called array elements. A symbolic name is used to refer to the array. Subscripts are used with array names to refer to individual array elements.
4. A reference to the name of a function followed by a list of arguments causes the computation indicated by the function definition to be performed. The resulting value is used in place of the function reference. Function references are treated in detail in Chapter 8.
5. An expression may be a single constant, variable, array element reference, or function reference, or it may be a combination of those components connected by operators, that specify computations to be performed on the values represented by those components to obtain a single result.

2.1 SYMBOLIC NAMES

Symbolic names are used to identify many entities within a FORTRAN program unit.

A symbolic name is a string of 1 to 6 letters and digits, the first of which must be a letter. Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(Begins with a digit)
K9	B.4	(Contains a special character)
X	X234567	(Contains more than 6 characters)

The following types of entities are identified by symbolic names:

Entity	Typed	Unique in Executable Program
Variables	yes	no
Arrays	yes	no
Arithmetic Statement Functions	yes	no
Processor-Defined Functions	yes	yes
Function subprograms	yes	yes
Subroutine subprograms	no	yes
Common blocks	no	yes
Block data subprograms	no	yes
Function entries	yes	yes
Subroutine entries	no	yes
External procedures	yes	yes

Within one program unit, the same symbolic name cannot be used to identify more than one entity. Within an executable program, comprised of more than one program unit, the same symbolic name can be used to identify only one of the entities indicated.

Each entity indicated as "Typed" in the above table has a data type. The means of specifying the data type of a name is discussed in Sections 5.1 and 5.2.

Within a subprogram, symbolic names are also used as dummy arguments. A dummy argument can represent a variable, array, array element, expression, or external procedure name.

2.2 DATA TYPES

Each basic component may represent data of one of several different types. The data type of a component may be inherent in its construction, implied by convention, implicitly declared, or explicitly declared. The data types available in FORTRAN, and their definitions, are as follows:

1. Integer - A whole number
2. Real - A decimal number; it may be a whole number with a decimal point, a decimal fraction, or a combination of the two
3. Double Precision - Similar to real, but with greater accuracy in its representation
4. Double Integer - Similar to integer but with a capacity for representing a greater number of significant digits
5. Logical - The logical value "true" or "false"
6. Hollerith - A string of 1 to 5 printing characters. The blank character is valid and significant in a Hollerith datum

An important attribute of each type of data is the amount of computer memory required to represent a value of that type.

The different data types require different amounts of storage in the XVM, the basic unit of which is an 18-bit word. Integer and Logical data use a single word each. Double Integers, Hollerith and Real data occupy two words each. Double Precision data requires three words of storage. (See Table 2-1).

Table 2-1
Data Type Storage Requirements

DATA TYPE	18-Bit Words
INTEGER	1
DOUBLE INTEGER	2
REAL	2
DOUBLE PRECISION	3
LOGICAL	1
HOLLERITH	2

Additional descriptions of these data types and their representations are presented in the sections that follow.

2.3 CONSTANTS

A constant represents a fixed value. A constant can represent a numeric value, a logical value, or a character string. There are six types of constants, each representing a different internal data format: INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION, LOGICAL and Hollerith.

2.3.1 Integer Constants

An integer constant is a whole number with no decimal point. It may be either a decimal or an octal number.

2.3.1.1 Decimal Integer Constants - The general form for a decimal integer constant is:

[±]nn

where

nn is a string of one to six decimal digits. Leading zeros, if any, are ignored.

A negative decimal integer constant must be preceded by a minus symbol; a positive constant may optionally be preceded by a plus symbol (an unsigned constant is presumed to be positive).

Except for a leading algebraic sign, a decimal integer constant cannot contain any character other than the numerals 0 through 9.

The value of a decimal integer constant cannot be greater than 131071 or less than -131072.

Examples

Valid Decimal Integer Constants	Invalid Decimal Integer Constants	
0	99999999999	(Too large)
-127	3.14	(Decimal point and comma not allowed)
+32123	32,767	
00555		

2.3.2 Octal Integer Constants

An octal integer constant is an alternate way of representing an integer constant. When used in an arithmetic context, octal constants are treated as decimal constants.

The general form for an octal integer constant is:

```
#nn
```

where

nn is a string of one to six octal digits.

Except for the leading pound sign, which must be present, an octal integer constant cannot contain any character other than the numerals 0 through 7.

An octal constant cannot be smaller than zero, nor greater than 777777.

Examples

Valid Octal Integer Constants	Invalid Octal Integer Constants	
#7213	32767	(Pound sign missing)
#1	#184	(Illegal character)
#17776	#3777777	(Number too large)
	#1	(Illegal character)

2.3.3 Double Integer Constants

A double integer constant is a whole number with no decimal point. It may be either a decimal or an octal number. It is similar to an integer constant but has the capacity for representing a greater number of significant digits.

2.3.3.1 Decimal Double Integer Constants - The general form for a decimal double integer constant is:

[±] nn

where nn is a string of six to eleven decimal digits. Fewer digits than this would make it a decimal integer constant (that is, a single integer). Leading zeros, if any, are ignored.

A negative decimal double integer constant must be preceded by a minus symbol; a positive constant may optionally be preceded by a plus symbol (an unsigned constant is presumed to be positive).

Except for a leading algebraic sign, a decimal double integer constant cannot contain any character other than the numerals 0 through 9.

Positive decimal double integer constants, n, must lie in the range $34,359,738,367 \geq n \geq 131,072$; negative decimal double integer constants must lie in the range $-34,359,738,368 \leq n \leq -131,073$ (otherwise, it is a decimal integer constant rather than a decimal double integer constant).

Examples

Valid Decimal Double Integer Constants	Invalid Decimal Double Integer Constants	
141520	400,000.00	(Decimal point and comma not allowed)
-34359738367	35000000000	(Too large)

2.3.3.2 Octal Double Integer Constant - The general form for an octal double integer constant is:

```
    #Dnn
```

where

nn is a string of one to twelve octal digits.

Except for the leading pound sign and letter D, which must be present, an octal double integer constant cannot contain any character other than the numerals 0 through 7.

An octal double integer constant cannot be smaller than zero nor greater than 777777777777.

Note that an octal integer value not preceded by a D may be assigned to a double integer variable, (i.e., DI = #130000). However, the magnitude of the integer must not exceed the limit for single integers (777777_8). If the magnitude is exceeded, its most significant digits will be truncated before it is assigned to the double integer variable.

Examples

Valid Octal Double Integer Constants	Invalid Octal Double Integer Constants	
#D0	D0	(# sign missing)
#D400000	#D776,377	(comma is illegal)
#D777777777777	#D1000000000000	(number too large)

2.3.4 Real Constants

A basic real constant is a string of decimal digits with a decimal point.

A basic real constant appears in one of the forms:

```
[±].nn      OR      [±]nn.nn      OR      [±]nn.
```

where nn is a string of numeric characters. The decimal point may appear anywhere in the string but following the sign symbol if one appears. The number of digits is not limited, but only the leftmost eight digits are significant. Thus, 12.345678 and 12.345679 are equivalent to 12.34567. Leading zeros (zeros to the left of the first non-zero digit) are ignored when counting the leftmost eight digits. Thus, in the constant 0.00001234567, all of the non-zero digits are significant.

A basic real constant must contain a decimal point.

A real constant may appear as a basic real constant, or as a basic real or decimal integer constant followed by a decimal exponent of the form:

```
E[±]nn
```

where nn is a 1- or 2-digit decimal integer constant. It represents a power of ten by which the preceding real or integer constant is to be multiplied (for example, 1E6 represents the value 1.0 x 10^6).

A minus symbol must appear between the letter E and a negative exponent; a plus symbol is optional for a positive exponent.

Except for algebraic signs, a decimal point, and the letter E (if used), a real constant cannot contain any character other than the numerals 0 through 9.

If the letter E appears in a real constant, a 1- or 2-digit integer constant must follow; the exponent field cannot be omitted, but may be zero.

A real constant cannot be greater in magnitude than 5.7896043×10^{76}, nor smaller in magnitude than $8.6361684 \times 10^{-78}$.

Examples

Valid Real Constants	Invalid Real Constants	
3.14159	1,234,567	(Commas not allowed)
621712.	325E-79	(Too small)
-.00127	-47.E81	(Too large)
+5.0E3	100	(Decimal point missing)
2E-3	$25.00	(Special character not allowed)
0.0		

2.3.5 Double Precision Constants

A double precision constant is a basic real constant, or an integer constant, followed by a decimal exponent of the form:

```
D[±]nn
```

where nn is a 1- or 2-digit decimal integer constant. The number of digits that precede the exponent is not limited, but only the leftmost 11 digits are significant.

A double precision constant is interpreted as a real number having a degree of precision somewhat higher than 10 significant eights.

A negative double precision constant must be preceded by a minus symbol; a positive constant may optionally be preceded by a plus symbol. Similarly, a minus symbol must appear between the letter D and a negative exponent; a plus symbol is optional for a positive exponent.

The exponent field following the letter D cannot be omitted, but may be zero.

The magnitude of a double precision constant cannot be smaller than $3.4359738367 \times 10^{-78}$, nor greater than $3.4359738367 \times 10^{76}$.

Examples

```
1234567890D+5
+2.718281828D00
-72.5D-15
  1D0
```

2.3.6 Logical Constants

A logical constant specifies a logical value, "true" or "false". Therefore, there are only two possible logical constants. They appear as:

```
    .TRUE.
```

and

```
    .FALSE.
```

The delimiting periods are part of each constant and must be present .

Internally, .TRUE. is given the integer value -1 (777777_8) and .FALSE. is given the value 0. Logical quantities may be operated on by either logical or arithmetic operators yielding, respectively, logical and arithmetic results.

2.3.7 Hollerith Constants

A Hollerith constant is a string of one to five printable ASCII characters preceded by a character count and the letter H.

Hollerith constants have the following general form:

$$nHc_1c_2c_3 \cdots c_n$$

where n is an unsigned integer constant from one to five stating the number of characters in the string (including spaces), and each c_i is a printable ASCII character.

Hollerith constants are stored as IOPS, ASCII 7-bit data five characters per two 18-bit words with the rightmost bit of the second word always zero.

A Hollerith constant, of the form nH chars, is stored as a real constant. A typical use of such constants is to test for user responses keyed in at the terminal.

```
    IF (A.EQ.3HYES)  GOTO 100
```

A must be a real variable in order to match it with the real Hollerith constant. Apart from this usage, Hollerith constants may be used only in CALL and DATA statements.

Examples:

Valid	Invalid
Hollerith Constants	Hollerith Constants
5HTODAY	3HABCD (Wrong number of characters)
1HΔ	HYES (Character count Missing)
5H$2.99	10HMESSAGEΔ12 (count>5)

2.3.7.1 Alphanumeric Literals - An alphanumeric literal is an alternate form of Hollerith constant.

The recommended form for an alphanumeric literal is:

'$c_1c_2c_3...c_n$'

where each c_i is a printable ASCII character. Both delimiting apostrophes must be present.

The rules for alphanumeric literals are similar to those for Hollerith constants, except that no character count is specified. The maximum number of characters in an alphanumeric literal is 5.

To represent the apostrophe character within an alphanumeric literal, write it as two consecutive apostrophes.

An alphanumeric literal, of the form 'chars', is stored internally as an unsigned double integer constant. A typical use of such constants is to test for user responses keyed in at a terminal.

```
    IF(J.EQ.'NO')  GOTO 200
```

J must be a double integer variable in order to match it with the double integer alphanumeric literal.

For historical reasons, the characters double quote and dollar sign can be used rather than single quote to enclose an alphanumeric literal, but single quote is the preferred character for compatibility with other DEC FORTRANs.

'ABC' and "ABC" and ABC are equivalent.

Examples

```
    'A'
    'A''BCD'      (Stored as A'BCD)
    'AΔB'         (Stored as AΔB. Blank not ignored)
```

2.4 VARIABLES

A variable is a symbolic name that is associated with a storage location. The size of the storage location, in terms of words of XVM memory, is a function of the data type of the variable. The value of the variable is the value currently stored in that location; that value can be changed by assigning a new value to that symbolic name. (The form of a symbolic name is given in Section 2.1).

Variables are classified by data type, just as are constants. The data type of a variable indicates the type of data it represents, its precision, and its storage requirements. When data of any type is assigned to a variable, it is converted, if necessary, to the data type of the variable. The data type of a variable may be established either by declaration or by implication.

Two or more variables are associated when each is associated with the same storage location; or, partially associated, when part (but not all) of the storage associated with one variable is the same as part or all of the storage associated with another variable. Association and partial association occur through the use of COMMON statements, EQUIVALENCE statements, and through the use of actual arguments and dummy arguments in subprogram references.

A variable is said to be defined if the storage location with which it is associated contains a datum of the same type as the name. A variable may be defined prior to program execution by means of a DATA statement or during execution by means of assignment or input statements.

If variables of differing types are associated (or partially associated) with the same storage location, then defining the value of one variable (for example, by assignment) causes the value of the other variable to become not defined.

2.4.1 Data Type by Definition

Data type declaration statements specify that given variables are to represent specified data types. For example, consider the following statements:

```
LOGICAL VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with a 1-word storage location that is to contain logical data, and that the variable VAR2 is to be associated with a 3-word double precision storage location. The explicit data typing statements, INTEGER, DOUBLE INTEGER, REAL, DOUBLE PRECISION and LOGICAL, are described in Section 5.2.

The IMPLICIT statement has a broader scope: it states that any variable having a name that begins with a specified letter, or any letter within a specified range, is to represent a specified data type, in the absence of an explicit type declaration. The IMPLICIT statement is explained in Section 5.1.

The data type of a variable may be explicitly specified only once. An explicit type specification takes precedence over the type implied by an IMPLICIT statement.

2.4.2 Data Type by Implication

In the absence of any IMPLICIT statements, all variables having names beginning with I, J, K, L, M, or N are presumed to represent integer data. Variables having names beginning with any other letter are presumed to be real variables. For example:

Real Variables	Integer Variables
ALPHA	KOUNT
BETA	ITEM
TOTAL	NTOTAL

2.4.3 Assigning Hollerith Data to Variables

The amount of Hollerith data that can be assigned to a variable depends on the data type of that variable. The maximum number of characters that can be stored for each data type is illustrated in Table 2-2.

Table 2-2
Hollerith Data Storage

VARIABLE TYPE	NUMBER OF CHARACTERS TO BE STORED
INTEGER	1 or 2
REAL	1 to 5
DOUBLE INTEGER	1 to 5

Hollerith data is stored in the IOPS ASCII form (5 7-bit characters in two words). If the number of characters stored is less than the maximum number for a particular type of variable, the FORTRAN system appends spaces to the end of the string to fill the variable to its capacity. In the case of INTEGER variables, the word contains 1 or 2 full characters, plus the high 4 bits of the representation for space. An attempt to assign more than the maximum number of characters causes the excess characters to be lost.

2.5 ARRAYS

An array is a group of contiguous storage locations associated with a single symbolic name, the array name. The individual storage locations, called array elements, are designated by subscripts appended to the array name. The number of subscripts required to locate an array element is the number of dimensions in the array.

An array may have from one to three dimensions. A simple column of figures is an example of a 1-dimensional array, requiring one subscript. To refer to a specific value in the column, say the ninth entry, we would simply request the ninth entry. If a page contained several columns of figures, that page might represent a 2-dimensional array, requiring two subscripts. To refer to a specific value in this array, we must locate it by both its entry (or row) number and its column number. If this table of figures covered several pages, we would have an example of a 3-dimensional array. To locate a value in this array, we would have to use its row number, its column number, and its page (or level) number.

The following FORTRAN statements establish arrays:

1. Data type declaration statement (Section 5.2),
2. DIMENSION statement (Section 5.3), and
3. COMMON statement (Section 5.4).

These statements, containing array declarators (array declarators are discussed in the following sub-section), define the name of the array, the number of dimensions in the array, and the number of elements in each dimension. The number of subscripts used thereafter to refer to a given array element must correspond to the number of dimensions defined by the array declarator for that array. (Subscripts are discussed in Section 2.4.4.)

2.5.1 Array Declarators

An array declarator specifies the symbolic name that identifies an array within a program unit and indicates the properties of that array.

An array declarator has the following form:

```
a (d[,d] ...)
```

a   is the symbolic name of the array -- the array name. (The form of a symbolic name is given in Section 2.1.)

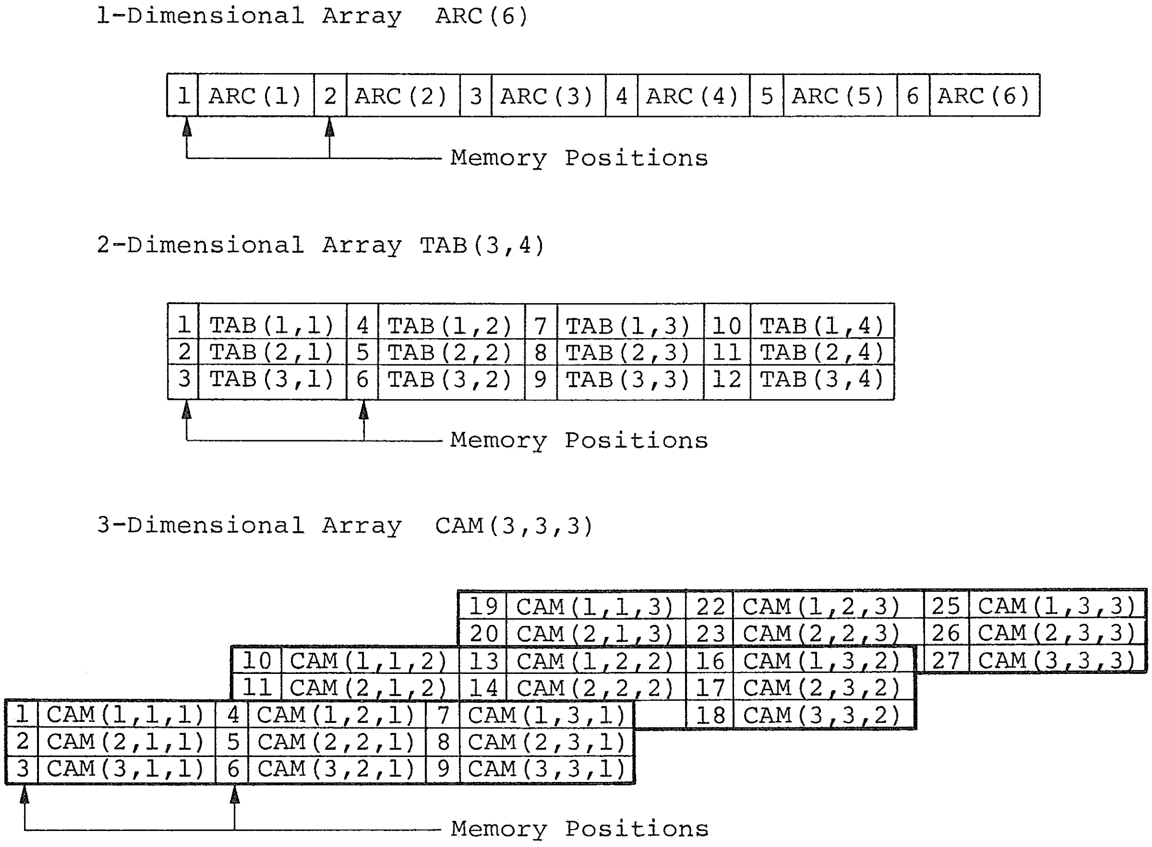
d   is the dimension declarator.

The number of dimension declarators indicates the number of dimensions in the array. The minimum number of dimensions is 1 and the maximum number is 3.

The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50 indicates that the dimension contains 50 elements. The dimension declarators may be constant or variable.

2.5.2 Array Storage

As discussed earlier in this section, it is convenient to think of the dimensions of an array as rows, columns, and levels or planes. However, the FORTRAN system always stores arrays in memory as a linear sequence of values. A 1-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multi-dimensional array is stored such that the leftmost subscripts vary most rapidly. This is called the "order of subscript progression". For example, consider the following array declarators and the arrays that they create:



1-Dimensional Array ARC(6)

1	ARC(1)	2	ARC(2)	3	ARC(3)	4	ARC(4)	5	ARC(5)	6	ARC(6)

2-Dimensional Array TAB(3,4)

1	TAB(1,1)	4	TAB(1,2)	7	TAB(1,3)	10	TAB(1,4)
2	TAB(2,1)	5	TAB(2,2)	8	TAB(2,3)	11	TAB(2,4)
3	TAB(3,1)	6	TAB(3,2)	9	TAB(3,3)	12	TAB(3,4)

3-Dimensional Array CAM(3,3,3)

19	CAM(1,1,3)	22	CAM(1,2,3)	25	CAM(1,3,3)
20	CAM(2,1,3)	23	CAM(2,2,3)	26	CAM(2,3,3)
21		24		27	CAM(3,3,3)

10	CAM(1,1,2)	13	CAM(1,2,2)	16	CAM(1,3,2)
11	CAM(2,1,2)	14	CAM(2,2,2)	17	CAM(2,3,2)
				18	CAM(3,3,2)

1	CAM(1,1,1)	4	CAM(1,2,1)	7	CAM(1,3,1)
2	CAM(2,1,1)	5	CAM(2,2,1)	8	CAM(2,3,1)
3	CAM(3,1,1)	6	CAM(3,2,1)	9	CAM(3,3,1)

Figure 2-1
Array Storage

2.5.3 Data Type of an Array

The Compiler establishes the data type of an array the same way it establishes data types for variables. In the absence of any data type specification, the data type of an array and its elements is implied by the initial letter of the array name. The data type may also be explicitly defined by data type declaration statements.

All of the values in an array are of the same data type. Any value assigned to any element of an array is converted to the data type of the array. If an array is named in a DOUBLE PRECISION statement, for example, the Compiler allocates a 3-word storage location for each element of the array. When a data value of any type is assigned to any element of that array, it is converted to double precision.

2.5.4 Subscripts

A subscript qualifies an array name. A subscript is a list of subscript expressions enclosed in parentheses and separated by commas that determines which element in the array is being or is to be referenced. The subscript is appended to the array name it qualifies. The terms "array element" and "subscripted variable" are synonymous.

A subscript has the following form:

```
(s[,s]...)
```

s   is a subscript expression

In any subscripted array reference, there must be one subscript expression for each dimension defined for that array (one subscript expression for each dimension declarator). For example, the following entry could be used to refer to the element located in the first row, third column, second level of the array CAM in Figure 2-1 (which is the element occupying memory position 16).

```
CAM (1,3,2)
```

Each subscript expression may be any valid integer-type arithmetic expression, provided that any array elements which appear in the expression have only single subscripts themselves.

In the following types of statements an array name may appear without a subscript:

Type declaration statements

COMMON

DATA statement

EQUIVALENCE statement

FUNCTION statement

SUBROUTINE statement

CALL statement

Input/Output statements

When one of these statements refers to an array name without subscripts, that statement specifies that either the entire array, or the first element of the array, depending upon the context, is to be used (or defined). The use of unsubscripted array names, in all other types of statements is prohibited.

In the EQUIVALENCE statement, a single subscript may follow the name of a multidimensional array. This usage is described in Section 5.5.

2.6 EXPRESSIONS

An expression represents a single value. It may be a single basic component, such as a constant or variable, or it may be a combination of basic components with one or more operators. Operators specify computations to be performed, using the values of the basic components, to obtain a single value.

Expressions may be classified as arithmetic, relational, or logical. Arithmetic expressions yield numeric values; relational and logical expressions produce logical values.

2.6.1 Arithmetic Expressions

Arithmetic expressions are formed with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic element may be any of the following:

1. A numeric constant
2. A numeric variable
3. A numeric array element
4. An arithmetic expression enclosed in parentheses
5. An arithmetic function reference (functions and function references are described in Chapter 8.)

The term "numeric" in these cases can also be interpreted to include logical data, since data of this type is treated as Integer data when used in an arithmetic context.

Arithmetic operators specify a computation to be performed using the values of arithmetic elements; they produce a numeric value as a result. The operators and their meanings are:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and Unary Plus
-	Subtraction and Unary Minus

The above are called binary operators, because each is used in conjunction with two elements. The - symbol may also be used as a "unary operator". When written immediately preceding an arithmetic element, a + or - denotes a positive or negative value.

Any arithmetic operator can be used in conjunction with any valid arithmetic element except for certain restrictions noted below.

A value should be assigned to a variable before its name is used in an arithmetic expression.

The following restrictions exist in regard to exponentiation ("No" indicates that a given combination is illegal):

FORTRAN Statement Components

BASE	EXPONENT			
	Integer	Double Integer	Real	Double
Integer	Yes	Yes	No	No
Double Integer	Yes	Yes	No	No
Real	Yes	Yes	Yes	Yes
Double	Yes	Yes	Yes	Yes

An element having a value of zero cannot be exponentiated by another zero-value element.

In any valid exponentiation, the result is of the same data type as the base element, except in the case of a real base and a double precision exponent (the result is double precision) and except in the case of an integer base and a double integer exponent (the result is double integer).

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. The precedence of the operators is as follows:

Operator	Precedence
**	First
Unary Minus	Second
* and /	Third
+ and -	Fourth

Whenever two or more operators of equal precedence (such as + and -) appear, they may be evaluated in any order chosen by the compiler so long as the actual order of evaluation is algebraically equivalent to a left to right order of evaluation. Exponentiation, however, is evaluated right to left. For example A**B**C is evaluated as A**(B**C).

Examples

```
-I+J/2*10+SQRT(A)**3
```

is evaluated as follows:

(1) the square root of A is computed and then raised to a power of 3,

(2) the value of I is negated,

(3) J is divided by 2 and then multiplied by 10,

(4) Finally, the partial results are added, first (-I) to (J/2*10), then to (SQRT(A)**3).

Another simple example, but one which illustrates the precedence operations at work and the need for caution is the raising of a negative number to some integer power.

```
-1**2 yields -1
```

This is not -1 raised to the power of 2; rather it is 1 raised to the power of 2 and then negated. To obtain the desired result, parentheses must be used:

```
(-1)**2 evaluates to 1
```

2.6.1.1 Use of Parentheses - Parentheses may be used to override the normal evaluation order. An expression enclosed in parentheses is treated as a single arithmetic element. That is, it is evaluated first to obtain its value, then that value is used in the evaluation of the remainder of the larger expression of which it is a part. An example of the effect of the use of parentheses is shown below (the numbers below the operators indicate the order in which the operations are performed).

```
4 + 3 * 2 - 6 / 2 = 7
  ↑   ↑   ↑   ↑
  3   1   4   2

(4+3) * 2 - 6 / 2 = 11
  ↑   ↑   ↑   ↑
  1   2   4   3
```

```
(4 + 3 * 2 - 6) / 2 = 2
    ↑   ↑   ↑    ↑
    2   1   3    4

((4+3) * 2 - 6) / 2 = 4
    ↑   ↑   ↑    ↑
    1   2   3    4
```

Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Nonessential parentheses, such as in the expression

```
4 + (3*2) - (6/2)
```

have no effect on the evaluation of the expression.

The use of parentheses to specify the evaluation order is often important in high accuracy numerical programs where evaluation orders that are algebraically equivalent might not be computationally equivalent when carried out on a computer.

2.6.1.2 Data Type of an Arithmetic Expression - If every element in an arithmetic expression is of the same data type, the value produced by the expression is also of that type. If elements of different data types are mixed together in an expression, the evaluation of that expression and the data type of the resulting value are dependent on a rank associated with each data type. The rank assigned to each data type is as follows:

Data Type	Rank
Integer	1 (Low)
Double Integer	2
Real	3
Double Precision	4 (High)

The data type of the value produced by an operation on two arithmetic elements of differing type is the same as that of the highest-ranked element in the operation. The data type of an expression is the same as the data type of the result of the last operation in that expression. The way in which the data type of an expression is determined is as follows:

1. Integer operations - Integer operations are performed only on integer elements. (When used in an arithmetic context, octal constants and logical entities are treated as integers.) In integer arithmetic, any fraction that may result from division is truncated, not rounded. For example, the value of the expression

   ```
   1/3 + 1/3 + 1/3
   ```

   is zero, not one.

2. Double Integer operations - Double integer operations are performed only on double integer elements or a combination of integer and double integer elements. Any integer elements present are converted to double integer. Then, double integer arithmetic is performed in the same manner as for integer-integer operations.

3. Real operations - Real operations are performed only on real elements or a combination of real, double integer and integer elements. Any integer and double integer elements present are converted to real type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. Note, however, that in the statement Y = (I/J)*X, an integer division operation is performed on I and J and a real multiplication is performed on the result and on X.

4. Double Precision operations - Any real, double integer or integer element in a double precision operation is converted to double precision type by making the existing element the most significant portion of a double precision datum; the least significant portion is zero. The expression is then evaluated in double precision arithmetic.

NOTE

The conversion of a real element to double precision does not increase its accuracy. For example, the real number 0.3333333 when converted becomes 0.33333330000 not 0.33333333333. Also note that real and double precision elements are only approximate representations of actual numbers. Values resulting from a real or double precision expression are only as accurate as the degree of precision for that data type.

2.6.2 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator. The value of the expression is either "true" or "false", depending on whether or not the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions. These operators are as follows:

Operator	Relationship
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The enclosing periods are part of each operator and must be present.

In a relational expression, the arithmetic expressions are evaluated first to obtain their values. Those values are then compared to determine if the relationship stated by the operator exists. For example, the expression:

```
     APPLE+PEACH  .GT.  PEAR+ORANGE
```

states the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If that relationship does in fact exist, the value of the expression is true; if not, the expression is false.

All relational operators have the same precedence. Thus, if two or more relational expressions appear within a logical expression (relational expressions are a subtype of logical expressions), the relational operators are evaluated from left to right. Arithmetic operators have a higher precedence than relational operators.

Parentheses may be used to alter the evaluation of the arithmetic expressions in a relational expression exactly as in any other arithmetic expression; but since arithmetic operators are evaluated before relational operators, it is unnecessary to enclose the entire arithmetic expression in parentheses.

When two expressions of different data types are compared by a relational expression, the value of the expression having the lower-ranked data type is converted to the higher-ranked data type before the comparison is made.

2.6.3 Logical Expressions

A logical expression may be a single logical element, or may be a combination of logical elements and logical operators. A logical expression yields a single logical value, true or false.

A logical element may be any of the following:

1. An Integer or Logical constant
2. An Integer or Logical variable
3. An Integer or Logical array element
4. A relational expression
5. A logical expression enclosed in parentheses
6. An Integer or Logical function reference (functions and function references are described in Chapter 8.)

The logical operators are shown below:

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction (logical AND). The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR). The expression is true if, and only if, either A or B, or both, is true.
.XOR.	A .XOR. B	Exclusive OR. The expression is true if A is true and B is false, or vice versa, but is false if both elements have the same value.

Operator	Example	Meaning
.NOT.	.NOT. A	Logical negation. The expression is true if, and only if, A is false.

When a logical operator is used to operate on logical elements, the resulting value is of type logical. When a logical operator is used with integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting value has type integer. When integer and logical values are combined with a logical operator, the operation is carried out as for two integer elements. The resulting type is integer.

A summary of all operators that may appear in a logical expression, and the order in which they are evaluated follows.

Operator	Evaluated
**	First
Unary Minus	Second
* and /	Third
+ and -	Fourth
.LT., .LE., .EQ., .NE., .GT., .GE.	Fifth
.NOT.	Sixth
.AND.	Seventh
.OR.	Eighth
.XOR.	Ninth

The delimiting periods of logical and relational operators must be present.

Operators of equal rank are evaluated from left to right. An example of the sequence in which a logical expression is evaluated is as follows:

```
    A*B+C*ABC .EQ. X*Y+DM*ZZ .AND. .NOT. K*B .GT. TT
```

is evaluated as:

```
(((A*B)+(C*ABC)).EQ.((X*Y)+(DM*ZZ))).AND.(.NOT.((K*B).GT.TT))
```

Parentheses may be used to alter the normal sequence of evaluation, just as in arithmetic expressions.

Two logical operators cannot appear contiguously (no intervening operand), except where the second operator is .NOT..

CHAPTER 3
ASSIGNMENT STATEMENTS

Assignment statements establish or alter the value of a variable or array element, by evaluating an expression and assigning the resulting value to the variable or array element. Variables may be reassigned any number of times within a program. Whenever references are made to these variables, their most recently assigned values are used.

Three types of assignment statements exist:

1. Arithmetic assignment statement
2. Logical assignment statement
3. ASSIGN statement

3.1 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement assigns the value of the expression to the right of the rightmost equal sign to the preceding elements going right to left. The previous values of the variables, if any, are lost.

The arithmetic assignment statement has the following form:

$v_1=v_2=...=v_n=e$

v_i is a numeric variable name or array element name.

e is an expression.

If an arithmetic assignment statement requires a continuation line, the equal sign (=) must appear on the first line. The equal sign does not mean "is equal to", as in mathematics. It means "is replaced by". Thus, the statement:

```
     KOUNT = KOUNT + 1
```

means, "Replace the current value of the integer variable KOUNT with the sum of that current value and the integer constant 1".

The effect of substituting values going right to left can be seen in this next example.

```
     TABLE(I) = I = 5
```

First the value of 5 is assigned to the variable I and then 5 is assigned to TABLE(5).

Although a symbolic name to the left of an equal sign may be initially undefined, values must have been previously assigned to all symbolic references in the expression.

If the data type of a variable or array element on the left of an equal sign is the same as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the entity on the left of the equal sign before it is assigned. A summary of data conversions on assignment is shown in Table 3-1. There are, however, situations in which the value obtained will be meaningless. For example, if the integer variable I is assigned the value of the double integer variable J, when J=100, the assignment will be as expected. When J=10000000, however, an unpredictable value assignment will result.

Table 3-1
Conversion Rules for Assignment Statements

	EXPRESSION (E)			
VARIABLE OR ARRAY ELEMENT (V)	INTEGER,LOGICAL, HOLLERITH OR CONSTANT	REAL	DOUBLE PRECISION	DOUBLE INTEGER
INTEGER	Assign E to V	Truncate E to Integer and assign to V	Truncate E to Integer and assign to V	Assign E to V
REAL	Append fraction (.0) to E and assign to V	Assign E to V	Assign MS portion of E to V; LS portion of E is rounded	Append a fraction of .0 to E and assign to V
DOUBLE PRECISION	Append fraction (.0) to E and assign to MS portion of V; LS portion of V is zero	Assign E to MS portion of V; LS portion of V is zero	Assign E to V	Append a fraction of .0 to E and assign to V
DOUBLE INTEGER	Assign E to LS portion of V. Propagate sign of E through MS portion of V	Truncate E to Double Integer and assign to V	Truncate E to Double Integer and assign to V	Assign E to V
LOGICAL	Assign E to V	Truncate E to Integer and assign to V	Truncate E to Integer and assign to V	Assign E to V

MS = Most Significant (high-order)

LS = Least Significant (low-order)

Assignment Statements

Examples

Valid Statements

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.

TABLE(I)=LIST(I+1)=0.
```

Invalid Statements

```
3.14 = A-B                  (Entity on the left must be a
                            variable or array element.)

-J = I**4                   (Entity on the left must not be
                            signed.)

ALPHA = ((X+6)*B*B/(X-Y)  (Invalid expression:  left and
                            right parentheses do not balance.)
```

3.1.1 Partword Notation

In addition to the basic arithmetic assignment statement, the programmer may use a part-word notation of the form:

```
[m : n]
```

where m and n are integer constants indicating a range from 0 to 35 $(0 \le m < n \le 35)$. This construction may optionally follow any variable, array element, or parenthesized expression in the expression portion of an arithmetic statement (to the right of =) and/or the variables or array elements being assigned. In the former case, the expression will be of type integer if $(n-m) \le 16$ and type double integer if $(n-m) \ge 17$; its value is bits m through n of the actual value (right adjusted). For example, the statement:

```
I=#2300[6:11]
```

assigns I the value 23_8, and

```
I=#2300[6:8]
```

assigns I the value 2. If I were a double integer, the statement

```
I=#2300[0:29]
```

would assign I the value 23. Note that #2300 is represented internally as 002300.

If this notation is used to the left of an equal sign, it indicates that only bits m through n of the variable are to be replaced by the value of the expression. For example, if the integer variable IVAR had previously been assigned the octal value 77, the statement:

```
IVAR[9:11]=#1
```

would make the new value of IVAR the octal integer 177. Only bit positions 9 through 11 are modified. Also, the statements:

```
IVAR=100
IVAR[9:11]=IVAR+1
```

leave the value of IVAR unchanged (i.e., 100). The programmer must be careful not to specify a double integer range (n>17) for an integer variable. For example:

```
I=#D77000000[19:35]
```

yields the single integer value 0.

Note that only the first two words of a double-precision floating variable (the exponent and first-order mantissa) may be manipulated via this notation.

3.2 LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement is similar to the arithmetic assignment statement, but operates with logical data. The logical assignment statement evaluates the expression on the right side of the rightmost equal sign and assigns the resulting logical value to the variable or array element on the left. When an integer element or subexpression appears in a logical assignment statement, if its value is non-zero, it is treated as .TRUE. and, if zero, as .FALSE..

The form of the logical assignment statement is shown below:

$$v_1 = v_2 = \ldots = v = e$$

v_i is a variable or array element of type Logical.

e is a logical expression.

The variables or array elements on the left of the rightmost equal sign must have been previously defined as being of logical type by a LOGICAL data type declaration statement or an IMPLICIT statement. Their values may be initially undefined.

Values, either numeric or logical, must have been previously assigned to all symbolic references that appear in the expression.

Examples

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

3.3 ASSIGN STATEMENT

The ASSIGN statement is used to associate a statement label with an integer variable. The variable may then be used as a transfer destination in a subsequent assigned GO TO statement (see Section 4.1.3) or an arithmetic IF statement (see Section 4.2.1).

The form of the ASSIGN statement is shown below:

```
ASSIGN s TO v
```

s   is a statement label of an executable statement in the same program unit as the ASSIGN statement.

v   is an integer variable.

The ASSIGN statement assigns the statement number to the variable in a manner similar to that of an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable. The variable must not be used as an integer before being redefined as an integer.

The ASSIGN statement must be executed before the assigned GO TO or arithmetic IF statement(s) in which the assigned variable is to be used. The ASSIGN statement, the assigned GO TO statement(s) and the arithmetic IF statement(s) must occur in the same program unit.

Assignment Statements

Consider the following example. In this example, the statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with statement 100. The constant 100 does not get assigned to NUMBER but rather the memory location associated with statement label 100. The statement

```
NUMBER = NUMBER+1
```

then becomes invalid, since it attempts to alter a statement label. This kind of error is not detectable by the FORTRAN system and can result in program failure. The statement:

```
NUMBER = 10
```

dissociates NUMBER from statement 100 and returns it to its status as an ordinary variable. It can no longer be used in an assigned GO TO statement, however.

Examples

```
ASSIGN 10 TO NSTART

ASSIGN 99999 to KSTOP

ASSIGN 250 TO ERROR      (ERROR must have been defined as an
                         integer variable.)
```

CHAPTER 4
CONTROL STATEMENTS

Statements are normally executed in the order in whicn they are written. However, it is frequently desirable to interrupt the normal program flow by transferring control ("branching" or "jumping") to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow, or may be based on a decision made at that point.

Transfer of control, whether within a program unit or to another program unit, is performed by control statements. These statements also govern repetitive processing ("looping") and program halts and waits. The various types of control statements are shown below:

```
GOTO
IF
DO
CONTINUE
CALL
RETURN
PAUSE
STOP
END
```

4.1 GO TO STATEMENTS

GO TO statements transfer control within a program unit, either to the same statement every time or to one of a set of statements, based on the value of an expression.

The three types of GO TO statements are:

1. Unconditional GO TO statement
2. Computed GO TO statement
3. Assigned GO TO statement

4.1.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the same statement every time it is executed.

The form of the unconditional GO TO statement is shown below:

```
        GO TO s
```

s   is the label of an executable statement in the same program unit.

The unconditional GO TO statement transfers control to the statement identified by the specified label. The statement label must identify an executable statement in the same program unit as the GO TO statement. Program execution continues from that point.

Examples

```
     GO TO 7734

     GO TO 99999

     GO TO 27.5        (Invalid; the statement label is improperly
                       formed.)
```

4.1.2 Computed GO TO Statement

The computed GO TO statement permits a choice of transfer destinations, based on the value of an integer variable within the statement.

The form of the computed GO TO statement is as follows:

```
        GO TO (slist) , v
```

slist   is a list of one or more executable statement labels separated by commas. The list of labels is called the transfer list. A maximum of 64 statement labels map appear in a computed GO TO list.

v   is an integer variable the value of which falls within the range 1 to n (where n is the number of statement labels in the transfer list).

The comma between the transfer list and the expression, the commas within the list, and the parentheses, are required.

The GO TO statement transfers control to the vth statement label in the transfer list. That is, if the list contains (30,20,30,40), and the value of v is 2, the GO TO statement passes control to statement 20, and so on.

If the value of the variable is less than 1, or greater than the number of labels in the transfer list, a warning error message is printed (.OTS 07) and control passes to the first executable statement following the computed GO TO.

Examples

```
     GO TO (12,24,36),INCHES

     GO TO (320,330,340,350,360),J
```

In the second example, if the value of the variable is 1, the GO TO statement transfers control to statement 320; if the value is 2, control passes to statement 330, and so on.

4.1.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to a statement label that is represented by a variable. Because the relationship between the variable and a specific statement label must be established by an ASSIGN statement, the transfer destination may be changed, depending upon which ASSIGN statement was most recently executed.

The assigned GO TO statement appears in the following form:

```
          GO TO v[,(slist)]
```

v   is an integer variable.

slist   (when present) is a list of one or more executable statement labels separated by commas.

The assigned GO TO statement transfers control to the statement whose label was most recently associated with the variable v by an ASSIGN statement.

The variable v must be of Integer type and must have been assigned a statement label by an ASSIGN statement (not an arithmetic assignment statement) prior to the execution of the GO TO statement.

The assigned GO TO statement and its associated ASSIGN statement(s) must exist in the same program unit. Statements to which control is transferred must also exist in the same program unit and must be executable.

Examples

```
GO TO INDEX

GO TO INDEX, (300,450,1000,25)
```

Note from the second example that statement labels in the transfer list need not be in ascending numeric order.

4.2 IF STATEMENTS

An IF statement causes a conditional control transfer or statement execution. There are two types of IF statements:

1. Arithmetic IF statement
2. Logical IF statement

In either type, the decision to transfer control or to execute the statement is based on the evaluation of an expression within the IF statement.

4.2.1 Arithmetic IF Statement

The arithmetic IF statement is used for conditional control transfers. It can transfer control to one of three statements, based on the value of an arithmetic expression.

The form of the arithmetic IF statement follows:

```
IF (e) V1, V2, V3
```

e   is an arithmetic or logical expression.

V   is a statement label or an ASSIGNed integer variable identifying an executable statement in the same program unit.

All three labels must be present. They need not refer to three different statements, however; they all may be the same. If desired, one or two labels may refer to the statement that immediately follows the IF statement. A transfer to that statement gives the effect of no transfer at all.

Each label may be either a statement label or an integer variable representing a label. The most recently assigned value of the variable must have been via an ASSIGN statement (not an assignment statement).

An arithmetic IF statement must appear entirely on one line. If this cannot be done, break the statement into two statements, one equating a variable to the arithmetic expression to be tested and the other an arithmetic IF statement which tests the value of that variable.

The arithmetic IF statement first evaluates the expression in parentheses and then transfers control to one of the three statement labels in the transfer list, as follows:

If the Value is:	Control Passes to:
Less than 0	Label V1
Equal to 0	Label V2
Greater than 0	Label V3

Examples

```
     IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI (giving a negative or zero value). Control passes to statement 100 only if THETA is greater than CHI.

```
     IF (NUMBER/2*2-NUMBER) 20,40,20
```

In this example, the IF statement transfers control to statement 40 if the value of the integer variable NUMBER is even and to statement 20 if it is odd (the fraction resulting from division by two is truncated, giving a lesser value when the quotient is re-multiplied). In this case, the third statement label is not used, since the expression can have only negative or zero values. The third label must be present, however.

```
     IF(A.GT.B.AND.A.GT.C) 3,4,5
```

In this example, the expression yields a logical result .TRUE. or .FALSE. . Since .TRUE. is internally represented as -1, if the expression is true, control will transfer to statement 3. Since .FALSE. is represented internally as 0, if the expression is false, control will pass to statement 4. Because only the values -1 or 0 can be generated, control will never transfer to statement 5.

```
     IF(I.XOR.J) 1,2,3
```

Logical expressions do not always yield -1 or 0 results. When a logical operator is used to operate on integer elements, the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting value is integer rather than logical. If I and J are integers, control will go to statement 2 if I=J, to statement 3 if I≠J but both have the same sign, or to statement 1 if their signs differ.

```
     IF (COUNT(3)) 100,J,100
```

If the value of the array element COUNT (3) is not zero, control goes to statement 100. If it is zero, control goes to the statement whose number was most recently assigned to J via an ASSIGN command.

4.2.2 Logical IF Statement

A logical IF statement causes a conditional statement execution. The decision to execute the statement is based on the value of a logical expression within the statement.

The form of the logical IF statement is:

```
          IF (e) st
```

e   is a logical expression.

st   is a complete FORTRAN statement. The statement cannot be a DO statement or another logical IF statement. (Any other executable statement is permitted.)

The logical IF statement first evaluates the expression. If the value of the expression is true, the IF statement causes the contained statement to be executed. If the value of the expression is false (zero), control passes immediately to the next executable statement following the logical IF.

Examples

```
IF (J .GT. 4 .OR. J .LT. 1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*(-1D0)

IF (ENDRUN) CALL EXIT

IF (X.EQ.3) IF (J) 3,55
```

In the last example, if X is not equal to 3.0, the arithmetic IF is executed and control either passes to statement 3 or statement 5 (otherwise control passes to the next executable statement below).

4.3 DO STATEMENT

DO statements are used to simplify the coding of iterative procedures. The DO statement causes the statements in its range to be repeatedly executed a specified number of times.

The DO statement appears in the following form:

```
DO s i=V1,V2[,[-]V3]
```

s — is the statement label of an executable statement. The statement must appear later in the same program unit.

i — is an integer variable.

V1,V2,V3 — are integer constants or integer variables.

The variable i is called the control variable of the DO and V1,V2,V3 are called the initial, terminal, and increment parameters respectively. If the increment parameter is omitted, a default increment value of 1 is used.

In the DO statement, an explicit minus sign may appear only preceding V3, the increment parameter. If integer constants are used for V1, the initial parameter, and V2, the terminal parameter, they must be positive or zero. Only V3 may appear as a negative integer constant. If integer variables are used for V1 or V2, they may take on negative, zero, or positive values. If an integer variable is used for V3, however, it can only have a positive and non-zero value. Negative increments are achieved by placing an explicit minus sign in front of the variable name. The absolute value of the difference between the values of the initial and the terminal parameters must be less than 131072. There is no code generated to test this condition at run time.

The statements that follow the DO statement, up to and including statement "s", are called the range of the DO loop. Statement "s" is called the terminal statement of the loop.

When the DO statement is encountered, the value of the initial parameter is assigned to the control variable. The executable statements in the range of the DO loop are then executed repeatedly for values of i starting with V1, incremented or decremented by V2 until i has surpassed the limit V2. This occurs when i is greater than V2 (for a positive increment) and when i is less than V2 (for a negative increment). The number of executions of the DO range (the iteration count) is given by

$$\left[\frac{V2 - V1}{V3}\right] \quad +1$$

where [X] represents the largest integer whose magnitude does not exceed the magnitude of X and whose sign is the same as that of X.

If the iteration count is zero, then the loop is executed once.

For each iteration of the DO loop, following execution of the terminal statement, the DO iteration control is executed:

1. The value of the increment parameter is algebraically added to the control variable.
2. If the iteration count is not exhausted, control returns to the first executable statement following the DO statement for another iteration of the range.

Exhaustion of the iteration count causes the normal termination of a DO loop and control to be passed to the next executable statement outside the range of the loop. The execution of a DO may also be terminated by a control statement within the range that transfers control outside the loop range. The control variable of the DO retains its current value if the loop is terminated in this way, but becomes undefined if the DO terminates normally.

If other DO loops share this same terminal statement, control is then passed to the next most enclosing DO loop in the nesting structure. In the case of the outermost DO loop in a nested structure, control is passed to the next executable statement following the terminal statement. If no other DO loops share the same terminal statement, control passes to the first executable statement following the terminal statement of the loop.

If the increment parameter is positive, the value of the terminal parameter must not be less than that of the initial parameter. Conversely, if the increment parameter is negative, the value of the terminal parameter must not be greater than that of the initial parameter. The value of the increment parameter must not be zero.

The terminal statement of a DO loop is identified by the label that appears in the DO statement. It must not be a GO TO statement of any type, an arithmetic IF statement, a RETURN statement, a STOP statement, a PAUSE statement or another DO statement. A logical IF statement is acceptable as the terminal statement, provided it does not contain any of the above statements.

The value of the control variable must not be altered within the range of the DO statement, nor should the values of the terminal and increment parameters. The control variable is available for reference as a variable within the range, however. (The control variable is frequently used as an array subscript to provide sequential manipulation of array elements.)

The range of a DO loop may contain other DO statements, as long as those "nested" DO loops conform to certain requirements (see Section 4.3.1).

Control may be transferred out of a DO loop, but cannot be transferred into a loop from elsewhere in the program. Exceptions to this rule are described in Section 4.3.3.

Examples

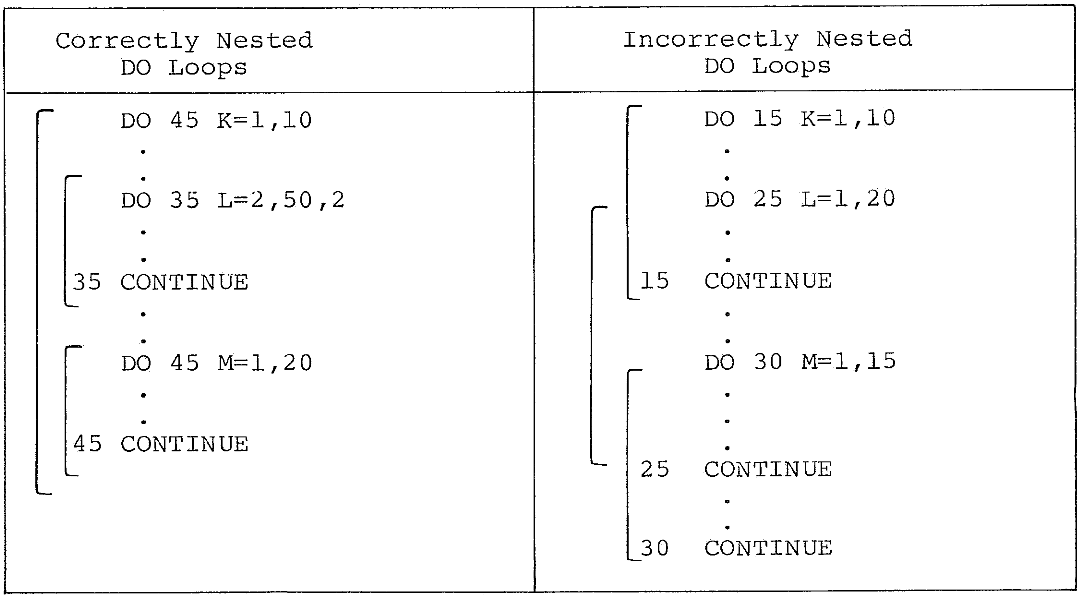
DO 100 K=1,50,2	(25 iterations, K=49 during final iteration)
DO 25 IVAR=1,5	(5 iterations, IVAR=5 during final iteration)
DO NUMBER=5,40,4	(invalid; statement label missing)
DO 40 M=2.10	(invalid; decimal point instead of comma)

The last example illustrates a common clerical error in that it is a valid arithmetic assignment statement in the FORTRAN language:

```
     DO40M = 2.10
```

4.3.1 Nested DO Loops

A DO loop may contain one or more complete DO loops. The range of an inner nested DO must lie completely within the range of the next outer loop. Nested loops may share tne same terminal statement. Nesting may occur to a depth of 9.



Correctly Nested DO Loops	Incorrectly Nested DO Loops
DO 45 K=1,10 . . DO 35 L=2,50,2 . . 35 CONTINUE . . DO 45 M=1,20 . . 45 CONTINUE	DO 15 K=1,10 . . DO 25 L=1,20 . . 15 CONTINUE . . DO 30 M=1,15 . . . 25 CONTINUE . . 30 CONTINUE

Figure 4-1
Nesting of DO Loops

Each time the outermost DO is executed for one value of its control variable, the DO within it is executed for all values of its control variable. In the left example of Figure 4-1, when the first DO is executed with K=1, the second DO is executed with L=2, L=4,...,L=50 (a total of 25 times) and then the third DO is executed with M=1, M=2,...,M=20 (a total of 20 times). Then the first DO is repeated, this time with K=2. Again, the second DO executes 25 times followed by the third DO, 20 times. When the first DO is completed, it will have executed 10 times; the second DO, 250 times; and the third DO, 200 times.

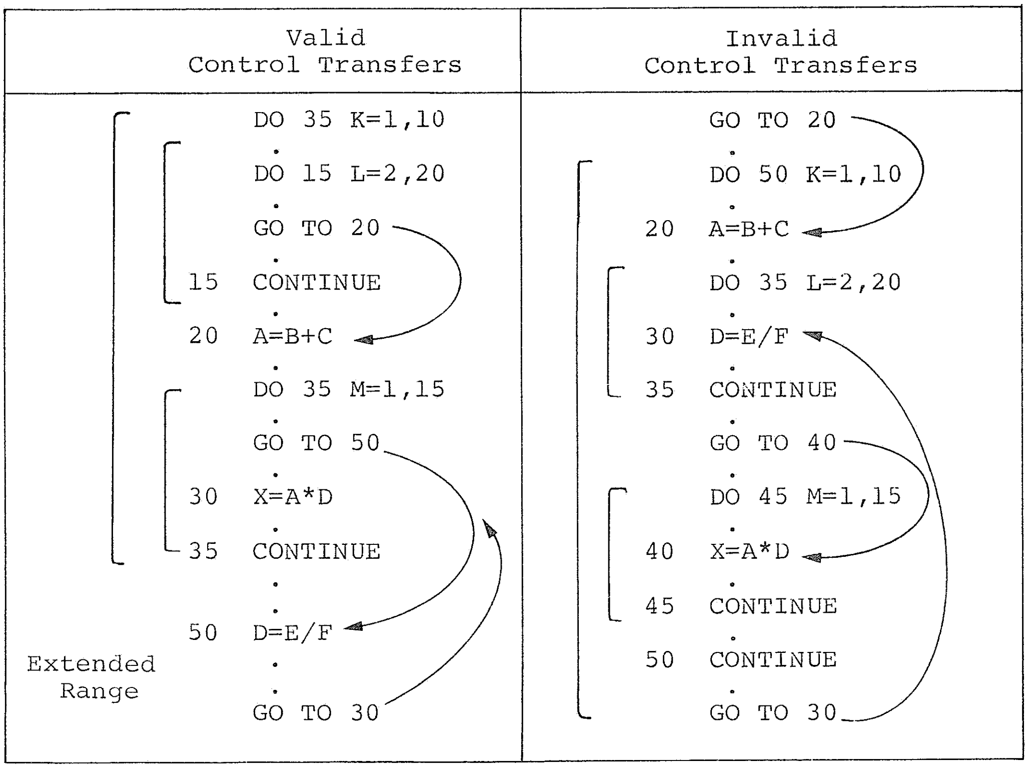
4.3.2 Control Transfers in DO Loops

As stated previously, control cannot be transferred into the range of a DO loop from outside that loop. However, within a DO loop, control may be passed from an inner loop to an outer loop. A transfer from an outer loop to an inner loop is prohibited. Exceptions to this rule are described in Section 4.3.3.

If two or more nested DO loops share the same terminal statement, control can be transferred to that statement only from within the range of the innermost loop. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

4.3.3 Extended Range

A DO loop is said to have an extended range if it contains a control statement that transfers control out of the loop and if, after the execution of one or more statements, another control statement returns control back into the loop. In this way the range of the loop is extended to include all of the executable statements between the destination statement of the first transfer and the statement that returns control to the loop.



Valid Control Transfers

```
        DO 35 K=1,10
        .
        DO 15 L=2,20
        .
        GO TO 20
        .
    15  CONTINUE
        .
    20  A=B+C
        .
        DO 35 M=1,15
        .
        GO TO 50
        .
    30  X=A*D
        .
    35  CONTINUE
        .
        .
    50  D=E/F
        .
        .
        GO TO 30
```

Invalid Control Transfers

```
        GO TO 20
        .
        DO 50 K=1,10
        .
    20  A=B+C
        .
        DO 35 L=2,20
        .
    30  D=E/F
        .
    35  CONTINUE
        .
        GO TO 40
        .
        DO 45 M=1,15
        .
    40  X=A*D
        .
    45  CONTINUE
        .
    50  CONTINUE
        .
        GO TO 30
```

Figure 4-2
Control Transfers and Extended Range

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

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

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

3. The extended range of a DO statement cannot change the control variable or parameters of the DO statement.

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

4.4 CONTINUE STATEMENT

The CONTINUE statement simply passes control to the next executable statement. It is used primarily as the terminal statement of a DO loop when that loop would otherwise end with a GO TO, arithmetic IF, or other prohibited control statement. If every DO loop ends with a

CONTINUE statement, the range of the loop is clearly visible in the program listing. (When used as the terminal statement of a DO loop or as the destination of a control transfer, it must be identified by a statement label.)

The form of the CONTINUE statement follows:

```
     CONTINUE
```

4.5 CALL STATEMENT

The CALL statement is used to transfer control from one program unit to another. It may also be used to transmit data between those program units.

The form of the CALL statement follows:

```
          CALL s[(a[,a]...)]
```

s is the name of a SUBROUTINE subprogram, a user-written Assembly Language routine, or a DEC-supplied system subroutine.

a is an actual argument by which data is transmitted between the calling program unit and the subroutine. Arguments to a subroutine are described fully in Section 8.1.3.

The CALL statement associates the data values in the argument list (if the list is present) with a matching set of dummy arguments in the subroutine, thereby making the data available to the subroutine, and transferring control to the subroutine to begin its execution.

The arguments in a CALL statement should agree in number, order, and data type with the dummy arguments in the subroutine definition. The CALL statement's arguments (called actual arguments) may be arithmetic expressions, Hollerith constants, array names, statement labels (preceded by @, as illustrated in Section 4.6), integer variables to which statement labels have been ASSIGNed, or external procedure names (if those names have been specified in an EXTERNAL statement, as described in Section 7.6). An unsubscripted array name in the argument list refers to the entire array. There is no practical limit on the number of arguments which may appear in a CALL statement.

Where user-written subroutines permit it, a CALL statement may contain more, or fewer, arguments than are specified in the subroutine definition, as long as an indication of some type (such as an argument that states how many other arguments are present) is given to the subroutine so that it will make no attempt to refer to a missing argument.

When subroutine execution has been completed (a RETURN statement has been encountered), control returns to the statement following the CALL.

Examples

```
CALL CURVE (BASE,3.14159,X,Y,LIMIT,RESULT)

CALL PNTOUT
```

4.6 RETURN STATEMENT

The RETURN statement is used to return control from a subprogram unit to the calling program unit.

The RETURN statement has the following form:

```
     RETURN [v]

v    is an integer variable
```

When a RETURN statement appears in a FUNCTION subprogram, it transfers control to the statement that contains the function reference (see Section 8.1.2) by which control was originally passed to the subprogram. When a simple RETURN statement appears in a SUBROUTINE subprogram, it returns control to the first executable statement following the CALL statement that transferred control to the subprogram.

A RETURN statement must not appear in a main program unit.

Example

```
      SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
      IF (N .LT. 10) GO TO 100
      DATA(K+2) = N-(N/10)*N
      N = N/10
      DATA(K+1) = N
      PRNT(K+2) = ALPH(DATA(K+2)+1)
      PRNT(K+1) = ALPH(DATA(K+1)+1)
      RETURN
100   PRNT(K+2) = ALPH(N+1)
      RETURN
      END
```

A RETURN statement can be used to return from a subroutine to a specified statement not immediately following the CALL. The CALL statement may pass statement labels as arguments by preceding them with an @, as in the following example:

```
     CALL COMPAR(A,B,@10,@20)
```

The use of multiple RETURN statements is illustrated in the subroutine below which compares two real variables and returns to different locations if they are equal, one is less than the other, or one is greater than the other.

```
     SUBROUTINE COMPAR(X,Y,LT,IGT)
     IF(X.EQ.Y) RETURN
     IF(X.LT.Y) RETURN LT
     RETURN IGT
     END
```

If the preceding CALL COMPAR is executed, control returns to the next executable statement following the CALL if A=B. Control returns to statement 10 if A<B. Control returns to statement 20 if A>B.

4.7 PAUSE STATEMENT

The PAUSE statement temporarily suspends program execution to permit some action on the part of the user.

The PAUSE statement appears in the following form:

```
          PAUSE  [disp]
     disp is an octal integer constant
```

The PAUSE statement prints the display (if one has been specified) at the user's terminal, suspends program execution, and waits for user response. When the user enters the appropriate control command, program execution resumes with the first executable statement following the PAUSE. Several PAUSE statements may appear in one program, and the numeric printout may be used to identify which PAUSE statement was reached. In XVM/RSX systems, the name of the task which executed the statement is included in the printout. To continue execution after the task has PAUSEd, in RSX use the operator command RESUME and in DOS type CTRL P.

Examples

```
PAUSE

PAUSE 77
```

4.8 STOP STATEMENT

The STOP statement is used to terminate program execution. Several such statements may appear in a program, and the numeric printout may be used to identify which termination point was reached.

The STOP statement appears in the following form:

```
STOP [disp]
```

disp is an octal integer constant.

The STOP statement prints the display (if one has been specified) at the user's terminal, terminates program execution, and returns control to the operating system. In XVM/RSX systems, the STOP message is printed only if the display argument is non-zero.

Examples

```
STOP

STOP 20
```

4.9 END STATEMENT

The END statement marks the physical end of a program unit. The END statement must be the last source line of every program unit.

The END statement has the following form:

```
END
```

In a main program, if control reaches the END statement, a CALL EXIT statement is implicitly executed; in a subprogram, a RETURN statement is implicitly executed.

CHAPTER 5
SPECIFICATION STATEMENTS

This chapter discusses the FORTRAN specification statements. Specification statements are nonexecutable. They provide the information necessary for the proper allocation and initialization of variables and arrays, and define other characteristics of the symbolic names used in the program, but have no function during the execution of the program. All specification statements must precede the executable portion of the program (see Figure 1-2).

5.1 IMPLICIT STATEMENT

The IMPLICIT statement permits the programmer to override the implied data type of symbolic names, in which all names that begin with the letters I, J, K, L, M, or N are presumed to represent integer data and all names beginning with any other letter are presumed to be of real type, in the absence of an explicit type declaration.

The IMPLICIT statement appears in the following form:

```
IMPLICIT typ(a[,a]...)[,typ(a[,a]...)]...
```

typ is one of the following data type names:

```
INTEGER
DOUBLE INTEGER
REAL
DOUBLE PRECISION
LOGICAL
```

and each a is an alphabetic specification in either of the following general forms:

```
a
```

or

```
a1-a2
```

a is an alphabetic character.

The latter form specifies a range of letters, from a1 through a2, which must occur in alphabetical order.

The IMPLICIT statement assigns the data storage and precision characteristics specified by "typ" to all symbolic names that begin with any specified letter, or any letter within a specified range. For example, the statements:

```
IMPLICIT INTEGER (I,J,K,L,M,N) or IMPLICIT INTEGER (I-N)
IMPLICIT REAL (A-H, O-Z)
```

represent the default in the absence of any data type specifications.

IMPLICIT statements must not be labeled.

Examples

```
IMPLICIT
IMPLICIT INTEGER (A,B, E-G), DOUBLE PRECISION D
```

In the above examples, since no statement has been made about the implicit mode of variables beginning with the letters C or H through Z, the default mode remains in effect for them. Only one IMPLICIT statement may appear in any one program or subprogram.

5.2 TYPE DECLARATION STATEMENTS

Type declaration statements explicitly define the data type of specified symbolic names.

Type declaration statements appear in the form shown below:

```
typ v[,v]...
```

typ is one of the following data type names:

LOGICAL
INTEGER
DOUBLE INTEGER
REAL
DOUBLE PRECISION

v is the symbolic name of a variable, array, or FUNCTION subprogram, or an array declarator.

A type declaration statement causes the specified symbolic names to have the specified data type.

A type declaration statement may also be used to define arrays, provided those arrays have not been previously defined, by including array declarators (see Section 2.4) in the list.

A type declaration overrides the data type implied by a symbolic name's initial letter, whether by default or by specification in an IMPLICIT statement.

Type declaration statements should precede all executable statements and all specification statements except the IMPLICIT statement. It must precede the first use of any symbolic name it defines.

The data type of a symbolic name may be explicitly declared only once.

Type declaration statements must not be labeled.

Examples

```
     INTEGER COUNT, MATRIX(4,4), SUM
     REAL MAN,IABS
     LOGICAL SWITCH

     DOUBLE PRECISION B,X,DA BS, DATAN
     B=DATAN(DABS(X))
```

The last two lines illustrate the need to declare arguments of functions to be of a particular mode as well as the functions themselves; the arguments do not take on the mode of the function by default, or vice versa.

5.3 DIMENSION STATEMENT

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

The form of the DIMENSION statement is:

```
          DIMENSION a(d)[,a(d)]...
```

a   is the symbolic name of an array

d   is a dimension declarator.

Each a(d) is an array declarator as described in Section 2.4.

The DIMENSION statement allocates a number of storage locations, one for each element in each dimension, to each array named in the statement. Each storage location is one, two, or three words in length, as determined by the data type of the array. The total number of locations assigned to an array is equal to the product of all dimension declarators in the array declarator for that array. For example:

```
     DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

defines ARRAY as having 16 real elements of two words each, and MATRIX as having 125 integer elements of one word each.

For further information concerning arrays and the storage of array elements, see Section 2.4.

The dimensions of an array may be defined by a type declaration statement. If an array has been so defined, it must not be redimensioned by a DIMENSION statement, and it must not be redefined by any other dimensioning statement.

Once the number of dimensions in an array has been defined, the same number of subscripts (or none) must appear in every reference to that array.

DIMENSION statements must not be labeled.

Examples

```
     DIMENSION BUD(12,24,10)

     DIMENSION X(5,5,5),Y(4,85),Z(100)
```

5.3.1 Adjustable Dimensions

The DIMENSION statement allows a subprogram to process more than one set of array data with a single definition through the use of integer variables as dimension declarators rather than unsigned integer constants. This facility is called Adjustable Dimensions.

To use adjustable dimensions, the user must first define one or more arrays explicitly in the main program unit. Then, when control is transferred to the subprogram containing the adjustable DIMENSION statement, the actual array name and the actual number of elements per dimension (for that execution of the subprogram) are passed to the subprogram as arguments in the function reference or CALL statement. The subprogram replaces the array name and adjustable dimensions in its DIMENSION statement(s) with those actual values to create the proper array definition for that execution. For example:

```
        Main Program                         Subprogram

  DIMENSION A(10), B(20)               SUBROUTINE SUB (X,N,R)
     .                                 DIMENSION X(N)
     .                                 DO 90 K=1,N
  CALL SUB (A,10,RESULT)                  .
     .                                    .
     .                                    .
  CALL SUB (B,20,ANSWER)                  .
     .                                 RETURN
     .                                 END
```

Each CALL statement in the main program supplies the subprogram with a different array name and number of elements, which are then associated with array name X and adjustable dimension N in the subprogram's DIMENSION statement. Thus, the subprogram processes a different set of data with each execution (note also that the value of N is used to determine the number of iterations of the DO loop).

Adjustable dimensions may only be used in subprograms; DIMENSION statements in the main program unit must use fixed dimension declarators.

Every call to a subprogram that contains an adjustable DIMENSION statement must pass an array name and actual dimension declarators as arguments to that subprogram. A dimension declarator passed as an argument is allowed to differ from the corresponding fixed dimension declarator for the array in question. However, the size of the adjustable array (the product of all dimensions) must not exceed the size of the array as declared when it was given fixed dimensions.

The value of a dummy argument which is used in an array declarator may be redefined during the execution of the subprogram. This does not affect the dimensions that are established upon entry to the subprogram.

5.4 COMMON STATEMENT

A COMMON statement defines one or more contiguous areas (blocks) of storage. Blocks are not necessarily contiguous with one another. Each block is identified by a symbolic name; in addition, one common block is also called the blank common block. A COMMON statement also defines the order of variables and arrays that are part of each common block.

Data in COMMON can be referenced from different program units by the same block name.

5.4.1 Blank Common and Named Common

There can be only one blank common block in an entire executable program. COMMON statements can be used to establish any number of named common blocks.

A COMMON statement has the following form:

```
          COMMON [/[cb]/] nlist [/[cb]/nlist]...

     cb   is a symbolic name (of the same form as a variable name),
          called a common block name, or is blank.  If the first cb
          is blank, the first pair of slashes may be omitted.

    nlist is a list of variable names, array names, and array
          declarators separated by commas.
```

A common block name may not be the same as a variable or array name; nor the same as the name of a function or a subroutine, or a function or subroutine entry, in the executable program.

Common blocks with the same name that are declared in different program units all share the same storage area when those program units are combined into an executable program.

Because assignment of components to common is on a one-for-one storage basis, components assigned by a COMMON statement in one program unit should agree in data type with those placed in common by another program unit. For example, if one program unit contains the statement:

```
     COMMON CENTS
```

and another program unit contains the statement:

```
     COMMON MONEY
```

unpredictable results may occur since the 1-word integer variable MONEY is made to correspond to the high-order word of the real variable CENTS.

The maximum size permitted for a COMMON block is 32,767 storage words, and this is also the maximum size for an array appearing in common. In XVM/DOS, the total available core memory space available for COMMON blocks is 128K words minus the size of the monitor, I/O routines and user program. This typically leaves about 110K of core, assuming that there is 128K of memory on the XVM and that XVM mode is enabled.

Example

```
         Main Program                        Subprogram

     COMMON HEAT,X/BLK1/KILO,Q          SUBROUTINE FIGURE
       .                                COMMON /BLK1/LIMA,R/ /ALFA,BET
       .                                  .
     CALL FIGURE                          .
       .                                  .
       .                                RETURN
       .                                END
```

The COMMON statement in the main program places HEAT and X in blank common and places KILO and Q in a labeled common block, BLK1. The COMMON statement in the subroutine causes ALFA and BET to correspond to HEAT and X in blank common and makes LIMA and R correspond to KILO and Q in BLK1.

All items to be stored in a given block need not be listed at once. For example,

```
     COMMON A,B/INTEGR/I,J,K//C,D.
```

This first assigns variables A and B to blank COMMON. Then I, J and K are assigned to the block named INTEGR. Finally, variables C and D are assigned to blank COMMON following A and B.

In general, programs which share COMMON blocks should declare them to be the same size as that used by all. Although there are situations where the XVM/DOS Linking Loader will permit COMMON sizes to vary, the exception rules are difficult to remember and are not recommended.

5.4.2 COMMON Statements with Array Declarators

Array declarators in the COMMON statement define the dimensions of an array in the same manner as a DIMENSION statement. Array names must not be otherwise subscripted (individual array elements cannot be assigned to common). A symbolic name that is intended to represent an array must be so defined at its first appearance in the program. It must not be redefined thereafter. Therefore, if an array has been defined in a DIMENSION or type declaration statement, it must not be redimensioned by a COMMON statement. Similarly, if an array is defined in a COMMON statement, it must not be subsequently redefined by any other dimensioning statement.

Example

```
     COMMON X(10,10),Y,Z(2)
```

5.4.3 COMMON Statements and Data Initialization

Elements of COMMON blocks may be initialized within BLOCK DATA subprograms. This can be done either using DATA statements or COMMON statements, as covered in Section 5.7.

5.5 EQUIVALENCE STATEMENT

The EQUIVALENCE statement declares two or more entities to be associated (either totally or partially) with the same storage location. The EQUIVALENCE statement works with components that exist in the same program unit.

The general form of the EQUIVALENCE statement is:

```
          EQUIVALENCE (nlist) [,(nlist)]...
```

nlist is a list of variables and array elements, separated by commas. At least two components must be present in each list. The array elements must have decimal integer constant subscripts.

The EQUIVALENCE statement causes all of the variables or array elements in one parenthesized list to be allocated beginning in the same storage location. Note that an Integer variable made equivalent to a Real variable shares storage with the high-order word of that variable. Mixing of data types in this way is permissible. Multiple components of one data type can share the storage of a single component of a higher-ranked data type. For example:

```
DOUBLE PRECISION DVAR
INTEGER IARR(3)
EQUIVALENCE (DVAR,IARR(1))
```

The EQUIVALENCE statement causes the three elements of the integer array IARR to occupy the same storage as the double precision variable DVAR.

The EQUIVALENCE statement can also be used to equate variable names. For example, the statement

```
EQUIVALENCE (FLTLEN, FLENTH, FLIGHT)
```

causes FLTLEN, FLENTH and FLIGHT to occupy the same storage provided they are also of the same data type. If not, then at least they all share the first word of data storage of the group.

An EQUIVALENCE statement in a subprogram must not contain dummy arguments.

Examples

```
EQUIVALENCE (A,B), (B,C)      (has the same effect as
                              EQUIVALENCE (A,B,C))

EQUIVALENCE (A(1),X), (A(2),Y), (A(3),Z)
```

5.5.1 Making Arrays Equivalent

When an element of an array is made equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the corresponding elements that are adjacent to those named in the statement. Thus, if the first elements of two equal-sized arrays are made equivalent, both entire arrays are made to share the same storage space. If the third element of a 5-element array is made equivalent to the first element of another array, the last three elements of the first array overlap the first three elements of the second array.

The EQUIVALENCE statement must not attempt to assign the same storage location to two or more elements of the same array, nor to assign memory locations in any way that is inconsistent with the normal linear storage of array elements (for example, making the first element of an array equivalent with the first element of another array, then attempting to set an equivalence between the second element of the first array and the sixth element of the other).

In the EQUIVALENCE statement only, it is possible to identify an array element with a single subscript, the linear element number, even though the array has been defined as a multi-dimensional array.

For example, the statements:

```
     DIMENSION TABLE (2,2), TRIPLE (2,2,2)
     EQUIVALENCE (TABLE(4), TRIPLE(7))
```

result in the entire array TABLE sharing a portion of the storage space allocated to array TRIPLE as illustrated in Figure 5-1.

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Figure 5-1
Equivalence of Array Storage

Figure 5-1 also illustrates that the statements

```
     EQUIVALENCE (TABLE(1),TRIPLE(4))
```

and

```
     EQUIVALENCE (TRIPLE(1,2,2), TABLE(4))
```

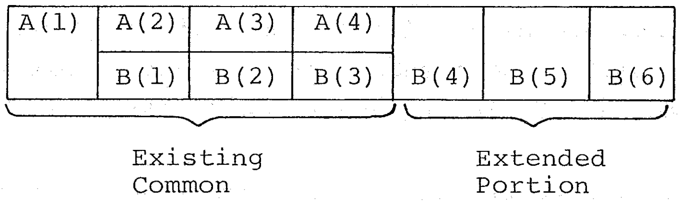
result in the same alignment of the two arrays. Note that the specification of an unsubscripted array name is equivalent to specifying the first element of the array.

5.5.2 EQUIVALENCE and COMMON Interaction

When components are made equivalent to entities stored in common, the common block may be extended beyond its original boundaries. An EQUIVALENCE statement can only extend common beyond the last element of the previously established common block. It must not attempt to increase the size of common in such a way as to place the extended portion before the first element of existing common. For example:

Valid Extension of Common

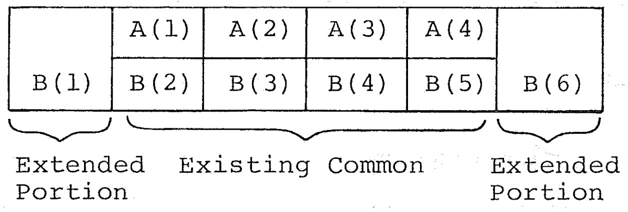
```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(1))
```



A(1)	A(2)	A(3)	A(4)			
	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)

Illegal Extension of Common

```
DIMENSION A(4),B(6)
COMMON A
EQUIVALENCE (A(2),B(3))
```



	A(1)	A(2)	A(3)	A(4)	
B(1)	B(2)	B(3)	B(4)	B(5)	B(6)

If two components are assigned to the same or different common blocks, they must not be made equivalent to each other.

All variables and array elements equivalenced to COMMON variables and COMMON array elements are treated as COMMON variables.

5.6 EXTERNAL STATEMENT

The EXTERNAL statement permits the use of external procedure names (functions, subroutines, and FORTRAN Library functions) as actual arguments to other subprograms.

The EXTERNAL statement appears in the following form:

```
          EXTERNAL v[,v]...

     v    is the symbolic name of a subprogram or the name of a
          dummy argument which is associated with a subprogram
          name.
```

The EXTERNAL statement declares each name in its list to be the name of an external procedure. Such a name may then appear as an actual argument to a subprogram. The subprogram may then use the associated dummy argument name in a function reference or a CALL statement. A subprogram uses dummy symbols in statements which obtain values when called; these values must all be defined in the calling program. If these values are program variables, they are already defined within the calling program. The EXTERNAL statement ensures that subprogram names are also defined. An arithmetic statement function, although classified as a subprogram, is not an external procedure and thus need not be referenced in an EXTERNAL statement.

Note, however, that a complete function reference used as an argument (such as CALL SUBR(A,SQRT(B),C), for example) represents a data value, not a subprogram name; if such is the case, the function name need not be defined in an EXTERNAL statement.

Example

```
         Main Program                     Subprograms

     EXTERNAL SIN,COS,TAN              SUBROUTINE TRIG (X,F,Y)
        .                              Y = F(X)
        .                              RETURN
     CALL TRIG (ANGLE,SIN,SINE)        END
        .
        .
     CALL TRIG (ANGLE,COS,COSINE)
        .
        .                              FUNCTION TAN (X)
     CALL TRIG (ANGLE,TAN,TANGNT)      TAN = SIN(X) / COS(X)
        .                              RETURN
        .                              END
```

The CALL statements pass the name of a function to the subroutine TRIG, which is subsequently invoked by the function reference F(X) in the second statement of TRIG. Thus, the second statement becomes in effect:

```
Y = SIN(X),
Y = COS(X), or
Y = TAN(X)
```

depending upon which CALL statement invoked TRIG (the functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN Library).

5.7 DATA STATEMENT

The DATA initialization statement permits the assignment of initial values to variables and array elements prior to program execution.

The DATA statement appears in the form:

```
DATA nlist/clist/[,nlist/clist/]...
```

nlist is a list of one or more variable names, array names, or array element names separated by commas. Subscript expressions must be integer constants. None of the elements are permitted to be dummy variables or arrays.

clist is a list of constants.

Constants in a clist may be written in either of the forms:

```
value
```

or

```
n * value
```

n is a nonzero unsigned integer constant that specifies the number of times the same value is to be assigned to successive entities in the associated nlist.

Value may be any type of constant. Double precision constants must be explicitly written in D format, such as 1.0D+01.

The DATA statement causes the constant values in each clist to be assigned to the entities in the preceding nlist. Values are assigned in a one-to-one manner in the order in which they appear, from left to right.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array. The associated constant list must therefore contain enough values to fill the array. Array elements are filled in the order of subscript progression.

When Hollerith data is assigned to a variable or array element, the number of characters that can be assigned depends on the data type of that component; a Hollerith constant that initializes an integer variable may not contain more than two characters. If the number of characters in a Hollerith constant is less than the capacity of the variable or array element, the constant is extended to the right with spaces. If the number of characters in the constant is greater than the maximum number that can be stored, the rightmost excess characters are not used.

The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names should also agree except in the case of Hollerith and integer constants. An integer constant may be assigned to a double integer variable or array element.

Example

```
     INTEGER A(10),BELL
     DATA A,BELL,STARS/10*0,7, '****'/
```

The DATA statement assigns zero to all ten elements of array A, the value 7 to the variable BELL, and four asterisks followed by a space to the real variable STARS.

5.7.1 Data Initialization of COMMON Elements

Elements within COMMON blocks may be initialized with data only within BLOCK DATA subprograms, which are described in Section 8.1.5. This can be done using DATA statements as described above or with a modified form of the COMMON statement which performs data initialization at the same time that it declares elements to be in COMMON.

Form:

```
    COMMON[/[cb]/] nlist [/[cb]/nlist]...;nlist/clist/[,nlist/clist/]...
```

cb is a symbolic name (of the same form as a variable name), called a common block name, or is blank. If the first cb is blank, the first pair of slashes may be omitted.

nlist is a list of variable names, array names, and array elements separated by commas.

clist is a list of constants.

The rules governing these elements are described for COMMON statements in Section 5.4 and for DATA statements in Section 5.7. The effect here is as if a COMMON and a DATA statement were concatenated on the same line, replacing the word "DATA" with a semicolon. Following the semicolon, only data initialization may appear.

Example

```
    COMMON/B/X,Y(5),I;X,I,Y(3)/2.0,3,5.0/
```

CHAPTER 6

INPUT/OUTPUT STATEMENTS

6.1 OVERVIEW

Input of data from external devices by a FORTRAN program is performed by READ statements. Output is performed by WRITE, TYPE, and PRINT statements. Some forms of these statements are used in conjunction with formatting information to translate and edit the data into a readable form.

Each READ or WRITE statement contains a reference to the logical unit to or from which data transfer is to take place. A logical unit can be connected to a device or file.

READ and WRITE statements fall into the following categories:

1. Unformatted Sequential I/O

   Unformatted sequential READ and WRITE statements transmit binary data in the sequence in which the data is physically stored on the device without translation.

2. Formatted Sequential I/O

   Formatted sequential READ and WRITE statements contain references to FORMAT statements, to format specifications in arrays, or use implied format specifications, that cause data to be translated to ASCII code on output, and to internal format on input.

3. Unformatted Direct Access I/O

   Unformatted direct access READ and WRITE statements perform input and output of binary data to and from direct access files. The files must have been defined by a CALL DEFINE statement. Direct access statements permit data to be transmitted to/from a device in random order, unrelated to the sequence in which the data is recorded on the device. Data may be changed in a single record without creating a new file.

4. Formatted Direct Access I/O

   Formatted direct access READ and WRITE statements contain references to FORMAT statements, or to format specifications in arrays, or use implied format specifications, and perform input and output of formatted data in direct access files.

Any type of READ statement can transfer control to another statement whenever an error condition or end-of-file condition is detected.

In addition to the above statements, the auxiliary I/O statements, REWIND and BACKSPACE, do not perform data transfer, but do file positioning functions. The ENDFILE statement writes a special form of record that will cause an end-of-file condition (an END= transfer) when read by an input statement. Finally, there are the ENCODE and DECODE statements, which perform data transfer and translation within memory.

The XVM operating systems have the ability to locate files by name on directoried devices. To extend this capability to the FORTRAN programmer, the following statements are provided.

1. CALL SEEK searches for a file of the specified name on the directoried device associated with the logical unit and then establishes a connection between the logical unit and that file. Once a file has been opened by CALL SEEK, formatted or unformatted sequential READs may be performed.

2. CALL ENTER initializes a file of the specified name on the directoried device associated with the logical unit. Once a CALL ENTER has been performed, formatted or unformatted data can be written sequentially into the file by WRITE statements.

3. CALL CLOSE terminates input from or output to a named file.

4. CALL RENAME allows the programmer to change the name of a named file on a directoried device.

5. CALL DELETE permits the programmer to delete a named file from a directoried device.

6. CALL FSTAT gives the programmer the ability to determine whether or not the named file is already present on the file is already present on the device associated with the logical unit number, without actually opening the file. This statement is relevant only to XVM/DOS systems, not XVM/RSX.

6.1.1 Input/Output Devices

FORTRAN uses the I/O services of the operating system and thus supports all peripheral devices that are supported by the operating system. I/O statements refer to the I/O devices by means of logical unit numbers. A logical unit number is an integer constant or variable with a positive value. Refer to the FORTRAN IV XVM Operating Environment Manual for additional information.

6.1.2 Format Specifiers

Format specifiers may be used in formatted I/O statements. A format specifier is either the statement label of a FORMAT statement or the name of an array containing Hollerith data interpretable as a format. Section 6 discusses FORMAT statements in detail.

6.1.3 Input/Output Records

Input/Output statements transmit all data in terms of records. The amount of information that can be contained in one record, and the way in which records are separated, depend on the medium involved.

For unformatted I/O, the amount of data to be transmitted is specified by the I/O statement. Unformatted data records may be arbitrarily large, restricted only by device size limitations. Internally, FORTRAN converts unformatted data records into a series of physical records which are linked together to form a single logical record. The size of the physical record depends upon the particular device. This is determined internally by FORTRAN and should not concern the FORTRAN programmer. Table 6-1 shows which standard peripheral devices can be used with unformatted I/O.

The amount of information to be transmitted by a formatted I/O statement is determined jointly by the I/O statement and specifications in the associated format specification. Formatted records are restricted in the amount of data they may contain. For the standard peripheral devices this is shown in Table 6-1. Unlike unformatted records, formatted records are not broken up into logically associated physical blocks.

The beginning of execution of an input or output statement initiates the transmission of a new data record. If an input statement requires only part of a record, the excess portion of the record is lost. In the case of formatted sequential input or output, one or more records can be transmitted with a single statement.

Table 6-1
Standard Devices I/O Limits

Device	Formatted ASCII Maximum Record Size (Characters)	Unformatted Binary Permitted?
Card Reader	80	No
DECtape	629	Yes
Disk	629	Yes
Line Printer	80 or 132	No
Magtape	629	Yes
Paper Tape	120	Yes
Teletypewriter	72	No

6.2 INPUT/OUTPUT LISTS

An I/O list specifies the data items to be manipulated by the statement containing the list. The I/O list of an input or output statement contains the names of variables, arrays, and array elements whose values are to be transmitted to or from a unit. An I/O list may be a single component or a series of such components, and it may contain an "implied DO" list, which specifies iterative transmission of values.

6.2.1 Simple Lists

A simple I/O list consists of a single variable, array name, or array element, or a series of such components separated by commas. The I/O statement assigns input values to, or outputs data from, the list elements in the order in which they appear, from left to right.

When an unsubscripted array name appears in an I/O list, a READ statement inputs enough data to fill every element of the array; a WRITE, TYPE, or PRINT statement outputs all of the values contained in the array. Data transmission begins with the initial element of the array and proceeds in the order of subscript progression, with the left-most subscript varying most rapidly. For example, if the unsubscripted name of a 2-dimensional array defined as:

```
ARRAY(3,3)
```

appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on, through ARRAY(3,3).

In a READ or ACCEPT statement, variables in the I/O list may be used as array subscripts elsewhere in the list. If, for example, the statement:

```
     READ (1,1250) J,K,ARRAY(J,K)
1250 FORMAT (I1,1X,I1,1X,F6.2)
```

was executed and the input record contained the values:

```
1,3,721.73
```

the value 721.73 would be assigned to ARRAY(1,3). The first input value is assigned to J and the second to K, thereby establishing the actual subscripts for ARRAY(J,K). Variables that are to be used as subscripts in this way must appear to the left of their use in the array reference.

6.2.2 Implied DO Lists

Implied DO lists are used to transmit only part of an array or to transmit elements in some sequence other than the order of subscript progression. This type of list element functions as though it were a part of an I/O statement that resides in a DO loop, and that uses the control variable or the imaginary DO statement to specify which data value or values are to be transmitted during each iteration of the loop.

An implied DO list appears as one or more data references followed by one or more control variable and parameter definitions, in the same form as that used in the DO statement. The data reference(s) and the first definition are enclosed in parentheses and separated by commas.

Each subsequent definition is separated from the preceding parenthesized set by a comma and enclosed in parentheses that also include all of the preceding entries. Implied DO parameters can be nested to a level of 9. For example:

```
WRITE (3,200) (A,B,C, I=1,3)

WRITE (6,15) ((P(I),Q(I,J), J=1,10), I=1,5)

READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

The first control variable definition is equivalent to the innermost DO of a set of nested loops, and therefore varies most rapidly. For example, the statement:

```
    WRITE (5,150) ((FORM(K,L), L=1,10), K=1,10,2)
150 FORMAT (F10.2)
```

is similar to:

```
    DO 50 K=1,10,2
    DO 50 L=1,10
    WRITE (5,150) FORM(K,L)
150 FORMAT (F10.2)
50  CONTINUE
```

Since the inner DO loop is executed ten times for each iteration of the outer loop, the second subscript, L, advances from one through ten for each increment of the first subscript. This is different from the normal order of subscript progression. Note also that since K is incremented by two, only the odd-number columns of the array will be output.

When multiple data references appear before the first control variable definition, data is transmitted to or from those references in the order in which they appear, before the incrementation of the first control variable. For example:

```
READ (3,999) (P(I),(Q(I,J), J=1,10), I=1,5)
```

assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ...  , Q(1,10),
P(2), Q(2,1), Q(2,2), ...  , Q(2,10),
 .      .       .             .
 .      .       .             .
P(5), Q(5,1), Q(5,2), ...  , Q(5,10)
```

When variables are output under control of an implied DO list, the values of those variables are repeatedly transmitted a number of times equal to the number of iterations of the implied DO loop. For example:

```
     WRITE (6,800) (A,B,C, I=1,3)
```

causes the values of the three variables to be output three times, in the order A, B, C, A, B, C, A, B, C.

When dealing with multidimensional arrays, it is possible to use a combination of fixed subscripts and subscripts that vary according to an implied DO. For example, the following statements:

```
     READ (3,5555) ((BOX(I,J), J=1,10), I=1,1)
     READ (3,5555) (BOX(1,J), J=1,10)
```

both have the same effect of assigning input values to BOX(1,1) through BOX(1,10), then terminating without affecting any other element in the first dimension of the array.

It is also possible to output the value of the implied DO's control variable directly, as in the statement:

```
     WRITE (6,1111) (I, I=1,20)
```

which simply prints the integers 1 through 20.

An implied DO list may be one element of a simple list.

The rules for the initial, terminal, and increment parameters, and for the control variable of an implied-DO list are the same as those for the DO statement (see Section 4.3).

6.3 UNFORMATTED SEQUENTIAL INPUT/OUTPUT

Unformatted input and output is the bit-for-bit transfer of binary data without conversion or editing. Unformatted I/O is generally used when data output by a program is to be subsequently input by the same program (or a similar program). Unformatted I/O saves execution time by eliminating the data conversion process, preserves greater accuracy in the external data, and usually conserves file storage space.

6.3.1 Unformatted Sequential READ Statement

The unformatted sequential READ statement initiates the input of a new record from the specified logical unit and assigns the data obtained to the components in the I/O list in the order in which they appear, from left to right. The amount of data each component receives is determined by its data type.

An unformatted sequential READ statement reads exactly one binary record. If the I/O list does not use all of the values in the record, the remainder of the record is discarded. If the contents of the record are exhausted before the I/O list is satisfied, an error condition results.

The unformatted sequential READ statement appears in the following form:

READ (u[,END=s_1] [,ERR-s_2]) [list]

u is a logical unit number (a positive integer constant or variable).

list is an I/O list.

s_1 is an executable statement label. The END and ERR options are explained in Section 6.7.

s_2 is an executable statement label.

If an unformatted sequential READ statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

The unformatted sequential READ statement must only be used to input records that were created by unformatted sequential WRITE statements.

Examples

READ (1) FIELD1, FIELD2	(Read one record from logical unit 1; assign input values to variables FIELD1 and FIELD2.)
READ (8)	(Advance through the file on logical unit 8 one record.)

6.3.2 Unformatted Sequential WRITE Statement

The unformatted sequential WRITE statement has the following form:

```
WRITE (u[,ERR=s]) list
```

u is a logical unit number (a positive integer constant or variable).

s is an executable statement label. The ERR option is explained in Section 6.7.

list is an I/O list.

PRINT and TYPE may be used as synonyms of WRITE.

The unformatted sequential WRITE statement outputs the values of the elements in the I/O list to the specified device in binary form, as one binary record. Therefore, the unit number must refer to a device capable of accepting binary data.

Examples

```
WRITE (1) (LIST(K),K=1,5)      (Outputs contents of elements 1
                               through 5 of array LIST to logical
                               unit 1.)
```

6.4 FORMATTED SEQUENTIAL INPUT/OUTPUT

Formatted input and output statements work in conjunction with FORMAT statements, format specifications stored in arrays, or implied format specifications to translate and edit data on output for ease of interpretation, and, on input, to convert data from external format to internal storage format.

6.4.1 Formatted Sequential READ Statement

The formatted sequential READ statement transfers data from the specified device and stores the input values in the elements of the I/O list in the order in which they appear, from left to right. At the same time, the format specifications referred to or implied by the READ statement translate the data from external to internal format.

The formatted sequential READ statement appears in the following form:

READ (u,[f][,END=s_1][,ERR=s_2])[list]

u is a logical unit number (a positive integer constant or variable).

f is a format specifier.

s_1 is an executable statement label. The END and ERR options are described in Section 6.7.

s_2 is an executable statement label.

list is an I/O list.

If the format specifier is omitted, data is read in free form (as described in Section 7.7) and converted to the modes of the variables in the I/O list. In such a case there is said to be an implied format. If both the format specifier and the I/O list are omitted, each READ statement will skip one input record.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or literal string, input data will be read and stored directly into the format specification. For example, the statements

```
    READ (5,100)
100 FORMAT (5H DATA)
```

cause five ASCII characters to be read from the terminal and stored in the Hollerith format descriptor. If the characters were HELLO, statement 100 would become:

```
100 FORMAT (5HHELLO)
```

If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of list elements exceeds the number of input fields, an error condition results unless the format specifications statement that one or more additional records are to be read (see Sections 6.4 and 6.8).

If no I/O list is present in a formatted sequential READ statement, the associated FORMAT statement or format array must contain at least one Hollerith field descriptor or alphanumeric literal to accept the input data. If it does not, no data transfer takes place; all data from the input record is lost and can only be retrieved by repositioning the file.

Examples

```
          READ (1,300) ARRAY                      (Reads record from
      300 FORMAT (20F8.2)                         logical unit 1,
                                                  assigns fields to
                                                  ARRAY.)

          READ (5,50)
      50  FORMAT (25H PAGE HEADING GOES HERE   )  (Reads 25 characters
                                                  from logical unit 5
                                                  and places them in
                                                  FORMAT statement.)

          DIMENSION FRMT(16)                      (Reads format
          READ (1,100) FRMT                       specification from
     100  FORMAT (16A5)                           logical unit 1 into
          READ (2,FRMT) A,B,C                     array FRMT and then
                                                  uses the array for-
                                                  mat to read data
                                                  from logical unit 2
                                                  into variables A,
                                                  B, and C.)

          READ(10,)A,J,Q                          (Reads ASCII data
                                                  from logical unit
                                                  10, converts it to
                                                  the mode of the
                                                  variables A,J, and
                                                  Q, and stores the
                                                  results in the
                                                  variables.)
```

6.4.2 Formatted Sequential WRITE Statement

The formatted sequential WRITE statement transmits the contents of the elements in the I/O list to the specified unit, translating and editing each value according to the format specifications referred to or implied by the WRITE statement.

The formatted sequential WRITE statement appears as:

```
WRITE (u,[f][,ERR=s])[list]
```

u   is a logical unit number (a positive integer constant or variable).

f   is a format specifier.

s   is an executable statement label. The ERR option is described in Section 6.7.

list is an I/O list.

PRINT and TYPE may be used as synonyms of WRITE.

If no I/O list is present, data transfer is entirely under the control of the format. The data to be output is taken from the format. In the case of implied format control, the result is a null record, that is, a line feed-Carriage RETURN sequence.

The data transmitted by a formatted sequential WRITE statement normally constitutes one formatted record. The FORMAT statement or format array may, however, specify that additional records are to be written during the execution of that same WRITE statement. In the absence of a format specifier, data in the I/O list is written in a fixed, implied format which is a function of data mode (see Section 7.7).

Numeric data output under format control is rounded during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. In this case, unformatted output is preferable to formatted output. Note also that unformatted data usually occupies less space on external devices than does formatted data.)

The records transmitted by a formatted WRITE statement must not exceed the length that can be accepted by the device. For example, a line printer typically cannot print a record that is longer than 132 characters.

Examples

```
          WRITE (6, 650)                (Outputs contents of FORMAT
     650  FORMAT (' HELLO, THERE')      statement to logical unit 6.)

          WRITE (1,95) AYE,BEE,CEE      (Writes one record of three
     95   FORMAT (F8.5,F8.5,F8.5)       fields to logical unit 1.)

          WRITE (1,950) AYE,BEE,CEE     (Writes three separate records
     950  FORMAT (F8.5)                 of one field each to logical
                                        unit 1.)

          WRITE(6,) A,B,C               (Writes out the variable names
                                        and values in the form:
                                             'name' = value.)
```

In the last example, format control arrives at the rightmost parenthesis of the FORMAT statement before all elements of the I/O list have been output. Each time this occurs, the current record is terminated and a new record is initiated. Thus, three separate records are written.

6.5 UNFORMATTED DIRECT ACCESS INPUT/OUTPUT

Unformatted direct access READ and WRITE statements are used to perform direct access I/O on any disk device. The CALL DEFINE statement is used to establish the number of records, and the size of each record, in a file to which direct access I/O is to be performed.

6.5.1 Unformatted Direct Access READ Statement

The unformatted direct access READ statement positions the input file to a record number and transfers the fields in that record to the elements in the data list in binary form without translation.

The unformatted direct access READ statement is written as follows:

```
          READ (u'r[,ERR=s]) [list]  ◄———————preferred form

       or

          READ (u#r[,ERR=s]) [list]
```

u   is a logical unit number (a positive integer constant or variable).

r   is an integer constant or variable that specifies the record number.

s is an executable statement label. The ERR option is described in Section 6.7.

list is an I/O list.

NOTE

In this form of READ statement an apostrophe is used to separate the logical unit number from the record number. For historical reasons the number sign character may be used in place of an apostrophe, but this is not standard notation.

If there are more fields in the input record than elements in the I/O list, the excess portion of the record is discarded. If there is no I/O list specified, the entire record is skipped over. If there is insufficient data in the record to satisfy the requirements of the I/O list, an error condition results.

The unit number in the unformatted direct access READ statement must refer to a file that has been opened for direct access.

The record number in an unformatted direct access READ statement must not be less than 1 nor greater than the number of records defined for the file, or an error condition results.

Examples

`READ (1'10) LIST(1),LIST(8)`	(Reads record 10 of a file on logical unit 1, and assigns two Integer values to specified elements of array LIST.)
`READ(4'58) (RHO(N),N=1,5)`	(Reads record 58 of a file on logical unit 4, and assigns five Real values to array RHO.)
`READ(2'I) A,B,C`	(Reads the I'th record of a file on logical unit 2, and assigns real values to A,B, and C.)

6.5.2 Unformatted Direct Access WRITE Statement

The unformatted direct access WRITE statement transmits the contents of the I/O list to a particular record number in a file on a directory-structured device. The data is recorded in binary form with no translation.

The unformatted direct access WRITE statement appears as follows:

```
        WRITE (u'r[,ERR=s]) list    ◄———— preferred form
    or
        WRITE (u#r[,ERR=s] list
```

u   is a logical unit number (a positive integer constant or variable).

r   is an integer expression that specifies the record number.

s   is an executable statement label. The ERR option is described in Section 6.7.

list   is an I/O list.

If the amount of data to be transmitted exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, the contents of the unused portion of the record are zero-filled.

For historical reasons, the pound sign character may be used in place of the apostrophe, but this is not standard notation.

PRINT and TYPE may be used as synonyms of WRITE.

Examples

WRITE (2'35) (NUM(K),K=1,10)	(Outputs ten Integer values to record 35 of a file on logical unit 2.)
WRITE (3'J) ARRAY	(Outputs the entire contents of ARRAY to a file on logical unit 3 into the record indicated by the value of J.)

6.5.3 CALL DEFINE Statement

The CALL DEFINE statement establishes the size and structure of a file upon which direct access I/O is to be performed. Although the method of creating and accessing data in direct access files is nearly identical in XVM/DOS and XVM/RSX systems, the physical structure of such files differs in the two systems. Therefore, a CALL DEFINE statement in XVM/DOS can create and alter only direct access files created within XVM/DOS, not those of RSX, and vice versa.

In XVM/DOS systems, the form of the statement is:

```
        CALL DEFINE (u,rs,nr,a,v,f,adj,d)
```

The format in XVM/RSX is the same, but there is an added optional argument:

```
        CALL DEFINE (u,rs,nr,a,v,f,adj,d[,ev])
```

u is a positive integer constant or integer variable that specifies the logical unit number.

rs is a positive integer constant or integer variable that specifies the length, in words or characters (for unformatted and formatted I/O respectively), of each record (the record size).

nr is an integer constant or integer variable in the range 1 to 131071 that specifies the number of records in the file.

a is the name of an array containing a six-character file name and a three-character extension, all in ASCII form. Alternatively, by specifying the integer constant or variable value of 0 instead of an array name, the system assigns a default temporary file name.

v is an integer variable, called the associated variable of the file, that, at the conclusion of each direct access I/O operation, contains the record number of the next sequential record in the file.

f is an integer variable or constant called the formatted/unformatted indicator. It is either 0 for unformatted (binary) or non-0 for formatted (ASCII). All records of each file must be of the same type, formatted or unformatted. Internally, IOPS I/O mode 0 will be used for unformatted files and IOPS I/O mode 2 will be used for formatted files.

adj is an integer constant or variable called the file SIZE adjustment parameter. In XVM/RSX forms of the CALL DEFINE statement, this argument is ignored (RSX does not permit file SIZE adjustment) but it must appear with some value. This parameter indicates, for a direct access file which was previously created by another CALL DEFINE, whether or not the size of the file is to be adjusted. If adj=0, no adjustment is performed. If adj≠0, the file size is changed to that of the nr parameter in the current CALL DEFINE statement.

d is an integer constant or variable which specifies whether or not to delete the temporary file when execution of this program has completed. A temporary file is declared by setting the a parameter in the CALL DEFINE statement to zero. Deletion is indicated if d≠0; if d=0, no deletion is performed.

ev is an optional parameter which appears only in XVM/RSX forms of CALL DEFINE. It is an integer event variable whose value on completion of the statement can be tested to see if any error occurred.

The CALL DEFINE statement specifies that a file containing nr fixed-length records of rs words (for unformatted I/O) or rs characters (for formatted I/O) each exists, or is to be created, on logical unit u. The records in the file are sequentially numbered from 1 through nr.

If the file already exists, the record size, rs, and the formatted/ unformatted indicator, f, of the file and the CALL DEFINE statement must match. In XVM/DOS only, if adj is zero, the number of records in the file must equal or exceed those of the CALL statement. If this is not the case, an error condition results and the statement is not executed. In XVM/DOS systems, this results in termination of program execution and an error message is printed on the console terminal. In XVM/RSX, program execution will continue. The user program can determine that there was an error by the fact that the value returned in the event variable, ev, is negative. The magnitude of the value indicates the specific cause of the error.

If no file name is specified in the CALL (a=0) a temporary file name of .TM0uu OTS is assumed; uu represents the logical unit u of the CALL statement as two decimal digits. If the explicitly named file in array a(a=0) or the temporary file .TM0uu OTS is not present on logical unit u, the file is created with the characteristics given in the CALL DEFINE. On completion of execution of the user program, the temporary file .TM0uu OTS may be either retained or deleted as a function of the d argument in CALL DEFINE.

In XVM/RSX, the initial contents of a direct access file are undefined. In XVM/DOS, however, the first time a direct access file is created, it is filled with records of zeroes (for unformatted files) or records of ASCII spaces (for formatted files), leaving the associated variable, v, set for record one.

The CALL DEFINE statement establishes the integer variable, v, as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in v the record number of the record immediately following the one just read or written. Since the associated variable always points to the next sequential record in the file (unless it is redefined by an assignment statement), direct access I/O statements can be used to perform sequential processing of the file, by using statements such as:

```
          READ (1'INDEX) ZETA,ETA,THETA
```

INDEX is the associated variable of the file in question.

If the file is to be processed by more than one program unit, or in an overlay environment, the associated variable should be placed in a resident named COMMON block.

In XVM/DOS systems only, when the size of an existing direct access file on unit u is to be adjusted, (adj≠0), the system creates a second, temporary file on logical unit u named ..TEMP OTS into which data from the original file is copied a record at a time. If the file is lengthened, the records in excess of the original file length are filled with spaces (for formatted records). Also, the associated variable v is set to one plus the original number of records. If the file size is being reduced, data from the end of the original file are lost and the associated variable is set to one. Once the adjusted file is created, the original file is deleted. Then the adjusted file is renamed to the name of the original file.

For unformatted direct access I/O, the record size may be as small as 1 and as large as 131071 binary words. For formatted direct access I/O, the record size may range from 1 to 629 ASCII characters. If the first character in the FORMAT statement is meant to be a vertical forms control character (the file may later be transferred to a printer) that character is counted as part of the record when defining the record size.

The programmer must be aware that the choice of record size has a bearing on disk space efficiency. Some guidelines will be given here.

Formatted records must fit entirely within a 256-word physical disk block, which the record size limit of 629 characters guarantees. Formatted records are always written as one record; so if the record size is slightly larger than half the capacity of the disk block, the remainder of the space in the block cannot be used and approximately half the disk space allocated for the file is unused.

Table 6-2 illustrates the effect on space efficiency of using different record sizes.

Table 6-2
Formatted ASCII Record Sizes

User Record Size (Characters)	Number of Records Per Physical Disk Block	Number of User Characters Per Block	Percentage of Maximum
629	1	629	100
330	1	330	52
309	2	618	98
204	3	612	97
149	4	596	95
119	5	595	95
99	6	594	94
84	7	588	93
69	8	552	88
64	9	576	92
54	10	540	86
49	11	539	86
44	12	528	84
39	14	546	87
34	15	510	81
29	18	522	83
24	21	504	80
19	25	475	76
14	31	434	69
9	42	378	60
8	42	336	53
7	42	294	47
6	42	252	40
5	42	210	33
4	63	252	40
3	63	189	30
2	63	126	20
1	63	63	10

One large record of 629 characters uses 100% of available disk storage whereas a record size of only one character uses only 10%. Do not attempt to interpolate data from values in this table. This is because the total number of characters per block which can be stored is a non-linear, saw-tooth function of user record size. Notice the linear improvement in efficiency going from a record size of one to a size of four. Going from four to five, there is a drop followed by another linear climb to a record size of nine. Each record size from nine up to 629 in the table is the peak of a sawtooth and represents a local maximum in space efficiency.

The maximum size for unformatted records is 131071 binary words. For INTEGER and LOGICAL data, which require one storage word per datum, the maximum number of data elements which can be transmitted in a single unformatted binary record is 131071. For REAL and for DOUBLE

INTEGER data, where two words per datum are needed, the maximum is 65535. For DOUBLE PRECISION data, which need three words per datum, the maximum is 43690. Remember however, that the record size, rs, specified in the CALL DEFINE statement is not a count of data elements; it is a count of the number of storage words used by those elements.

The operating system requires the size of a physical record to be a multiple of two. Since the FORTRAN system imposes an overhead of three words for unformatted record identification, the user must specify an odd record size. If an even record size is used, the number will be treated as if a number one larger were specified.

For record sizes smaller than or equal to 251 words, which means a record can fit entirely within one physical block, multiple records can be packed in a single block but a record is not broken up to reside part in one block and the remainder in another. For record sizes larger than 251 words, the converse is true. A large record is broken up into smaller components, completely filling all blocks except, possibly, the last. However, the next record will begin at the start of the next physical block so that record packing is not performed.

Table 6-3 illustrates space efficiency for binary records which fit in a single disk block in the same way as Table 6-2 does for formatted records. A record size of 251 makes maximum use of disk space whereas a size of one is only one fourth as good. Do not attempt to interpolate values from Table 6-3. The number of data words per disk block is a non-linear, sawtooth function of the user record size in the same way as illustrated in Table 6-2 for formatted records. For example, when going from a record size of 123 to 124, the efficiency drops from 98 percent to 49 percent because only one unformatted binary record of size 124 (remember the three-word overhead) can fit in a 256-word disk block.

For records which span more than a single physical block, the space utilization can be either good or bad. Record sizes which are a multiple of 251 make maximum use of file storage. Record sizes which are a multiple of 251 plus one require an extra storage block of 256 words for the one extra word.

Table 6-3
Unformatted Binary Record Size for Single Block Records

User Record Size (Binary Words)	Number of Records per Physical Disk Block	Number of Data Words Per Disk Block	Percentage of Maximum Per Block
251	1	251	100
249	1	249	99
247	1	247	98
245	1	245	98
.	.	.	.
.	.	.	.
.	.	.	.
123	2	246	98
81	3	243	97
59	4	236	94
47	5	235	94
39	6	234	93
33	7	231	92
27	8	216	86
25	9	225	90
21	10	210	84
19	11	209	83
17	12	204	81
15	14	210	84
13	15	198	79
11	18	195	77
9	21	189	75
7	25	175	70
5	31	155	62
3	42	126	50
1	63	63	25

The CALL DEFINE statement must be executed before the first direct access I/O statement that refers to the specified file. Once I/O to the file is completed, the file should be closed by one of the following statements (where a is the logical unit number).

```
ENDFILE u
```

or

```
CALL CLOSE (u)
```

Examples

```
CALL DEFINE (15,99,100,0,NUM,1,0,1)
```

This specifies that logical unit 15 is to refer to a formatted ASCII file of 100 fixed-length records, each record of which is 99 characters long. The records are numbered sequentially from 1 through 100.

Initially, the value of the associated integer variable NUM is set to 1. After each direct access I/O operation on this file, NUM will contain the number of the record immediately following the one just processed. Since the array name argument is zero, a temporary file named .TM015 OTS is created. After completion of program execution, the temporary file will be deleted.

```
     DIMENSION ANAME(2)
     DATA ANAME/'RANDO', 'MFIL'/

     CALL DEFINE(12,302,500,ANAME,NUM,0,1,0)
```

This specifies that logical unit 12 is to refer to an unformatted binary file whose size is to be adjusted so that it is 500 records long. [In XVM/RSX systems, because the file size adjustment parameter is ignored, if the file already exists it must contain 500 records.] The adjusted file is written to logical unit 12 and replaces the original file. Each fixed-length record contains 302 binary words. The records are numbered sequentially from 1 through 500. Initially, the value of the associated integer variable NUM is set to 1. After each direct access I/O operation on this file, NUM will contain the number of the record immediately following the one just processed. The name of the file is RANDOM FIL and is stored in ASCII form in the real array called ANAME.

6.6 FORMATTED DIRECT ACCESS INPUT/OUTPUT

Formatted input and output statements work in conjunction with FORMAT statements, format specifications stored in arrays, or implied format specifications to translate and edit data on output for ease of interpretation, and, on input, to control data from external format to internal storage format.

6.6.1 Formatted Direct Access READ Statement

The formatted direct access READ statement causes the specified record to be read from the direct access file currently connected to the unit. The information in the record is scanned and converted as specified by the referenced or implied format specification. The resulting values are assigned to the elements specified by the list.

The statement has the following form:

```
     READ (u'r,[f][,ERR=s])[list] ◄—preferred form
  or
     READ (u#r,[f][,ERR=s])[list]
```

u   is a logical unit number (a positive integer constant or variable).

r   is an integer expression that specifies the record number.

f   is a format specifier.

s   is an executable statement label. The ERR option is described in Section 6.7.

list   is an I/O list.

NOTE

In this form of READ statement an apostrophe is used to separate the logical unit number from the record number. For historical reasons, the pound sign character may be used in place of an apostrophe, but this is not standard notation.

If the format specifier is omitted, data is read in free form (as described in Section 7.7) and converted to the modes of the variables in the I/O list. In such a case there is said to be an implied format.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or literal string, input data will be read and stored directly into the format specification. For example, the statements

```
      READ (5'3, 100)
 100  FORMAT (5H DATA)
```

cause five ASCII characters to be read from record 3 and stored in the Hollerith format descriptor. If the characters were HELLO, statement 100 would become:

```
 100  FORMAT (5HHELLO)
```

If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of list elements exceeds the number of input fields, an error condition results unless the format specifications state that one or more additional records are to be read (see Sections 7.4 and 7.8).

If no I/O list is present in a formatted direct access READ statement, the associated FORMAT statement or format array must contain at least one Hollerith field descriptor or alphanumeric literal to accept the input data. If it does not, no data transfer takes place.

If the list and format specification require more characters than a record contains, all of the list elements become undefined and an error condition exists.

6.6.2 Formatted Direct Access WRITE Statement

The formatted direct access WRITE statement writes the specified record in the direct access file that is currently connected to the unit. The list specifies a sequence of values which are converted to characters and positioned as specified by the referenced or implied format specification.

The statement has the following form:

```
        WRITE (u'r,[f][,ERR=s])[list]  ◄──── preferred form

     or

        WRITE (u#r,[f][,ERR=s])[list]
```

u   is a logical unit number (a positive integer constant or variable).

r   is an integer expression that specifies the record number.

f   is a format specifier.

s   is an executable statement label. The ERR option is described in Section 6.7.

list   is an I/O list.

For historical reasons, the pound sign character may be used in place of an apostrophe, but this is not standard notation.

PRINT and TYPE may be used as synonyms of WRITE.

If no I/O is present, data transfer is entirely under the control of the format. The data to be output is taken from the format. In the case of implied format control, the result is a null record, that is, a line feed-carriage return sequence.

The data transmitted by a formatted sequential WRITE statement normally constitutes one formatted record. The FORMAT statement or format array may, however, specify that additional records are to be written during the execution of that same WRITE statement. In the absence of a format specifier, data in the I/O list are written in a fixed, implied format which is a function of data type (see Section 7.7).

Numeric data output under format control is rounded during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. In this case, unformatted output is preferable to formatted output. Note also that unformatted data usually occupies less space on external devices than does formatted data.)

If the values specified by the list and format do not fill the record, blank characters are appended to fill the record.

If the list and format identifiers specify more characters than can fit into a record, an error condition exists.

6.7 TRANSFER OF CONTROL ON END-OF-FILE OR ERROR CONDITIONS

Any type of READ or WRITE statement may contain a specification that control is to be transferred to another statement if the I/O statement encounters an error condition or (for READ only) the end of the file. These specifications appear as follows:

END=s_1

and

ERR=s_2

s_1,s_2 are the statement labels of executable statements to which control is to be transferred.

A READ statement may contain either or both of the above specifications, in either order. A WRITE statement may only specify the ERR option. Any such specification must follow the unit number, record number, and/or format specification.

If an end-of-file condition is encountered during an I/O operation, the READ statement transfers control to the statement named in the END=s specification. If no such specification is present, an error condition results.

If a READ or WRITE statement encounters an error condition during an I/O operation, it transfers control to the statement whose label appears in the ERR=s specification. If no ERR=s specification is present, the I/O error causes the program execution to terminate.

The statement label that appears in the END=s or ERR=s specification must refer to an executable statement that exists within the same program unit as the I/O statement.

Examples of I/O statements containing END=s and ERR=s specifications follow:

READ (8,END=550) (MATRIX(K),K=1,100)	(Passes control to statement 550 when end-of-file is encountered on logical unit 8.)
WRITE (5,50,ERR=390)	(Passes control to statement 390 on error.)
READ (1'INDEX,ERR=150) ARRAY	(Passes control to statement 150 on error.)

NOTE

An end-of-file condition can not occur during direct access READ statements. An END=s specification may be included in direct access READ statements; however, transfer of control to the label will never occur. In particular, attempting to READ or WRITE a record using a record number greater than the maximum specified for the unit is an error condition. If these errors occur and the options to detect them do not appear in the READ/WRITE statements, program execution will be terminated and an error message will be printed.

In XVM/DOS, once a branch is made to a statement label specified by the ERR= option, the type of error can be determined by invoking an integer function, IOERR (N). The value of the function, i.e.,

```
IVAL=IOERR (M)
```

will be one of the following:

Value	Error Indicated
-1	Parity error
-2	Checksum error
-3	Record-too-large error
-5	End-of-file detected
-6	End-of-medium detected
+N	Where N represents the code number of a recoverable OTS type error (see Appendix D for a list of OTS error codes)

6.8 AUXILIARY INPUT/OUTPUT STATEMENTS

The statements in this category are used to perform file management functions. REWIND, BACKSPACE and ENDFILE are normally associated with unnamed files; the other statements, with named files.

6.8.1 REWIND Statement

The form of the REWIND statement follows:

```
REWIND u
```

u is a logical unit number (a positive integer constant or variable).

For non-directoried magtape or DECtape[1], where files are unnamed and multiple files can exist on one unit, the REWIND command repositions the tape to its initial (load) point preceding the first file in the

[1] The non-directoried mode of operation on DECtape is a capability of XVM/DOS systems not available in XVM/RSX.

sequence. This type of tape operation (in XVM/DOS but not XVM/RSX systems) is simulated on disk (or on directoried DECtape and magtape) whenever a sequential READ or WRITE statement is not preceded by a CALL DEFINE, CALL SEEK or CALL ENTER. A file named .TM0uu OTS is opened on the disk (where uu is the ASCII representation of logical unit u as a decimal number) when READ or WRITE is executed in tnis manner; REWIND to the disk simply closes the file. If the file was opened with a WRITE statement, prior to closing the file, REWIND writes an end-of-file record.

Example

```
    REWIND 3        (Repositions logical unit 3 to beginning of
                    currently open DOS disk file.)
```

6.8.2 BACKSPACE Statement

The BACKSPACE statement repositions a currently open file backward one record and repositions to the beginning of that record. On the execution of the next I/O statement, that record is available for processing.

The BACKSPACE statement is written as follows:

```
          BACKSPACE u
```

u is a logical unit number (a positive integer constant or variable).

If the magnetic tape is at load point, (or in XVM/DOS systems only) if DECtape is at block zero or if at the beginning of the disk file, the BACKSPACE statement is ignored. For disk, BACKSPACE is allowed only for input files.

Example

```
    BACKSPACE 4     (Repositions open file on logical unit 4 to
                    beginning of the previous record.)
```

6.8.3 ENDFILE Statement

On non-directoried magtape or DECtape, the ENDFILE statement writes an end-of-file record on a currently open sequential file.

The ENDFILE statement is written as follows:

```
        ENDFILE u
```

u is a logical unit number (a positive integer constant or variable).

The tape is positioned beyond the end-of-file record so that additional files can be written. On directoried magtape[1] or DECtape, the ENDFILE statement writes an end-of-file record on a currently open sequential file and closes the file, causing its name and attributes to be recorded in the tape's file directory. On disk, ENDFILE writes an end-of-file record for a currently open sequential file which was opened either by CALL ENTER or (in XVM/DOS only) by WRITE. In all cases where a file is open on disk, ENDFILE closes the file. Unlike operation on magtape where multiple files can be written separated by end-of-file records, only one file per logical disk unit can be written in this fashion.

Example

```
    ENDFILE 2
```

The above statement writes an end-of-file record (if necessary as explained above) to the currently open file on logical unit two.

6.8.4 CALL SEEK Statement

The CALL SEEK statement is used to locate a named, sequential access or DOS direct access file on a directoried device and to open the file for subsequent sequential READs.

Direct access files in RSX are formatted differently from sequential access files and cannot be opened by a CALL SEEK statement. The file is not typed as formatted or unformatted in the CALL SEEK statement; instead, this is specified in each READ statement and must correspond with the mode in which the file was written.

[1] The non-directoried mode of operation on DECtape and the directoried mode on magtape are capabilities of XVM/DOS systems not available in XVM/RSX.

6.8.4.1 CALL SEEK Statement in XVM/DOS - The DOS form of the CALL SEEK statement is written as follows:

```
CALL SEEK (u,a)
```

where:

u   is a logical unit number (a positive integer constant or variable)

a   is the name of a real or double integer array containing the file name.

If a file is already open on unit u when CALL SEEK is executed, or if the file specified in array a does not exist on that unit, program execution will terminate and a system error message will appear on the console terminal.

The filename and extension are formed from 7-bit ASCII characters and are stored left justified in the array. The filename occupies the first six characters in the array, the extension occupies the next three, and the remaining character is a blank.

DOS
Example

```
DIMENSION FILEN(2)
DATA FILEN/5HTEMP ,4H BIN/
CALL SEEK (1,FILEN)
```

This will open a file called TEMP BIN for sequential READs on logical unit 1.

6.8.4.2 CALL SEEK Statement in XVM/RSX - The RSX form of the CALL SEEK statement is written as follows:

```
CALL SEEK (u,nH name, nH ext[,ev])
```

u   is a logical unit number (a positive integer constant or variable).

n   is a number of characters (an unsigned decimal integer constant) in "name" and in "ext".

name   is a string of from one to five ASCII characters representing the file name.

ext is a string of from one to three ASCII characters representing the file name extension.

ev is an optional integer event variable.

If the named file existing on unit u is a sequential file, and is not being used in a conflicting way (for instance, it is illegal to read such a file if it is at the same time being written), the file is opened to permit data input by subsequent sequential READ statements. If not, an error condition exists, no file is opened, and a negative value is returned in the event variable, ev, to indicate the type of error. If the CALL SEEK is performed sucessfully, a positive value is returned in the event variable. Control will return to the user program before completion of the I/O request; this condition is indicated by a value of zero in the event variable.

RSX
Example

```
     CALL SEEK (15,4HTEMP,3HBIN,IEV)
10   CALL WAITFR (IEV)
     IF(IEV) 999, 10, 20

20
```

In this example, a file named TEMP BIN is sought in the directory of the device on logical unit 15. Control returns to statement 10 before this operation is finished; if IEV is zero, indicating that the SEEK operation has not completed, control remains at statement 10. The WAITFR subroutine relinquishes control to the lower priority tasks in the XVM/RSX system until IEV becomes non-zero. If an error has occurred, control goes to statement 999. Otherwise, normal processing continues at statement 20.

6.8.5 CALL ENTER Statement

The CALL ENTER statement is used to create or replace a named sequential access or DOS direct access file on a directoried device and to open the file for subsequent sequential WRITEs.

Direct access files written in RSX are formatted differently from sequential access files and cannot be opened by a CALL ENTER statement. The file is not typed as formatted or unformatted in the CALL

ENTER statement; instead, this is specified in each WRITE statement. A file may contain formatted or unformatted data, but not both.

6.8.5.1 CALL ENTER Statement in XVM/DOS - The DOS form of the CALL ENTER statement is written as follows:

```
          CALL ENTER (u,a)

     u    is a logical unit number (a positive integer constant
          or variable).

     a    is the name of a real or double integer array containing
          the file name
```

If a file is already open on unit u when CALL ENTER is executed, or the file exists on logical unit u as an RSX-type of direct access file (which cannot be opened by a CALL ENTER statement), program execution will terminate and a system error message will appear on the console terminal. The filename and extension are formed as explained in Section 6.8.4.1 on CALL SEEK in XVM/DOS.

If the file named in the CALL ENTER statement already exists in the directory of the device connected to logical unit u and is a sequential access or DOS direct access file, it will be deleted as soon as the new file is closed. Direct access files written in DOS are structured identically with sequential access files with the constraint that the record size remain constant throughout the file. Normally direct access files are created or adjusted by using the CALL DEFINE statement and CALL ENTER is used for sequential access files not meant to be treated as direct access too.

DOS
Example

```
     DIMENSION FILEN(2)
     DATA FILEN/5HRANDO, 4HMNUM/
     CALL ENTER(3,FILEN)
```

This will open a file named RANDOM NUM on logical unit 3 for subsequent sequential WRITEs.

6.8.5.2 CALL ENTER Statement in XVM/RSX - The RSX form of the CALL ENTER statement is written as follows:

```
          CALL ENTER (u,nH name,nH ext[,ev])
```

u is a logical unit number (a positive integer constant or variable).

n is the number of characters (an unsigned decimal integer constant) in "name" and in "ext".

name is a string of from one to five ASCII characters representing the file name.

ext is a string of from one to three ASCII characters representing the file name extension.

ev is an optional integer event variable

While a file is being initialized for output with a CALL ENTER statement in RSX, control will return to the calling program before completion of the request. This is indicated by the fact that the event variable, assuming one is specified in the call, has a zero value. A positive and non-zero value indicates successful completion; a negative value indicates an error.

If a file is already open on logical unit u or if the named file exists as an RSX random access file, the command will not be executed. A negative number will be stored in the event variable, ev, to indicate the source of the error.

When the named file already exists on logical unit u (and is not an RSX direct access file), it will be deleted as soon as the new file is closed. Direct access files created in DOS are structured identically with sequential access files, but are constrained to a constant record size within a given file. Direct access files are normally created using the CALL DEFINE statement; and sequential access files, with CALL ENTER.

RSX
Example

```
          CALL ENTER (15,5HRANDM,3HNUM,IEV)
     50   CALL WAITFR(IEV)
          IF (IEV) 999,50,51

     51
```

Here, a file named RANDM NUM is opened on logical unit 15. Statement 51 relinquishes control to lower priority tasks until the event variable becomes non-zero. An error causes a branch to statement 999; whereas, normal completion continues at statement 51.

6.8.6 CALL CLOSE Statement

The CALL CLOSE statement indicates the completion of a set of related I/O operations on the currently open file. It is used primarily to directoried devices to close files opened via CALL SEEK or CALL ENTER (or CALL RENAME in RSX systems). CALL CLOSE, when used without specifying a filename as explained below for RSX, is functionally equivalent to the ENDFILE statement described in Section 6.8.3.

6.8.6.1 CALL CLOSE Statement in XVM/DOS - The DOS form of the CALL CLOSE statement is:

```
          CALL CLOSE (u)
```

u — is a logical unit number (a positive integer constant or variable)

DOS
Example

```
          CALL CLOSE (5)
```

This statement closes the file on logical unit 5, writing an end-of-file record as necessary.

6.8.6.2 CALL CLOSE Statement in XVM/RSX - In RSX the CALL CLOSE statement is written:

```
          CALL CLOSE (u[,nHname,nHext[,ev]])
```

u — is a logical unit number (a positive integer constant or variable

n — is the number of characters (an unsigned decimal integer constant) in "name" and in "ext".

name — is a string of from one to five ASCII characters representing the file name

ext — is a string of from one to three ASCII characters representing the file name extension.

ev — is an optional integer event variable.

The filename and extension are optional. If they are specified and a file was opened by a CALL RENAME statement, the file will appear in the device's directory under the new name (if unique) and the old file name will be erased. If the new name is not unique, the renaming process will not take place and the event variable will be set with a negative value to indicate this type of error.

If the file name and extension are specified but the original file was not opened by a CALL RENAME command, they are ignored.

When a file name and extension are not specified, the opened file is closed with the original name.

If a CALL CLOSE is executed but no file is open, an appropriate negative value is set in the event variable.

Successful closing of an open file will be indicated by a positive and non-zero value in the event variable.

RSX
Examples

```
          CALL CLOSE (15)
```

This statement will close the file which was open on logical unit 15.

```
          CALL RENAME (15,4HTEST,3H000,IEV)
     30   CALL WAITFR (IEV)
          IF (IEV) 999,30,31
     31   CALL CLOSE (15,4HTEST,3H001,IEV)
     32   CALL WAITFR (IEV)
          IF (IEV) 999,32,33
```

This sequence of statements causes the version number of file TEST, stored in the file name extension, to be incremented by 1. Statements 30 and 32 relinquish control to lower priority tasks in RSX until the event variable becomes non-zero. Then execution goes either to an error processor (statement 999) or to the next statement in sequence.

6.8.7 CALL FSTAT Statement

The CALL FSTAT statement, which is available in XVM/DOS but not XVM/RSX, is used to detect the presence or absence of a named file on a given logical unit.

6.8.7.1 CALL FSTAT Statement in XVM/DOS - The CALL FSTAT statement is written as:

```
CALL FSTAT (u, a, v)
```

u   is a logical unit number (a positive integer constant or variable).

a   is the name of a real or double integer array containing the file name.

v   is an integer variable whose value is set, on completion of the statement, to indicate presence of the file (v=1) or lack thereof (v=0).

CALL FSTAT, if issued while a file is already open on logical unit u, will cause program execution to terminate and a system error message to appear on the console terminal. CALL FSTAT is used to determine whether or not a file exists without actually opening the file.

The file name and extension are formed as explained in Section 6.8.4.1 on CALL SEEK in XVM/DOS.

DOS
Example

```
     DIMENSION F(2)
     DATA F/5HINPUT,4H 001/
     CALL FSTAT (2,F,I)
     IF(I.EQ.0) GO TO 999
     CALL SEEK (2,F)
```

In this example, first a check is made to see if file INPUT 001 exists. If not, a branch is taken to an error processor. If it does exist, the file is opened for reading by the CALL SEEK statement.

6.8.8 CALL RENAME Statement

The CALL RENAM (XVM/DOS form), CALL RENAME (XVM/RSX form) statement is used to change the name of a file on a directoried device associated with the given logical unit number.

6.8.8.1 CALL RENAM Statement in XVM/DOS - In XVM/DOS, the format of the CALL RENAM statement is:

```
          CALL RENAM (u,al,a2,v)
```

u — is a logical unit number (a positive integer constant or variable).

al,a2 — are the names of real or double integer arrays which contain, respectively, the current file name and the new file name.

v — is an integer variable whose value is set, on completion of the statement, to indicate presence of the current file (v=1) or lack thereof (v=0).

CALL RENAM, if issued while a file is already open on logical unit u, will cause program execution to terminate and a system error message to appear on the console terminal. On completion of the statement, success is indicated if the variable v is returned a value of 1. If not, v is set to zero.

The file names and extensions are formed as described in Section 6.8.4.1 on CALL SEEK in XVM/DOS.

DOS
Example

```
     DIMENSION FILOLD(2), FILNEW(2)
     DATA FILOLD/5HOLDNA,4HMSRC/
     DATA FILNEW/5HNEWNA,4HMSRC/
     CALL RENAM (1,FILOLD, FILNEW, I)
```

This example illustrates how a file on logical unit 1 named OLDNAM SRC is changed to NEWNAM SRC.

6.8.8.2 CALL RENAME Statement in XVM/RSX - In XVM/RSX, the format of the CALL RENAME statement is:

```
          CALL RENAME (u,nHname,nHext[,ev])
```

u — is a logical unit number (a positive integer constant or variable).

n — is the number of characters (an unsigned decimal integer constant) in "name" and in "ext".

name — is a string of from one to five ASCII characters representing the file name.

ext is a string of from one to three ASCII characters representing the file name extension.

ev is an optional integer event variable.

In RSX, the CALL RENAME statement opens a file for the sole purpose of changing its name. The file is closed and the new name is specified in the CALL CLOSE statement.

CALL RENAME, if issued while a file is already open on logical unit u, or if issued for a file which doesn't exist on that unit, will cause a negative value to be returned in the event variable indicating these errors. If the file is successfully opened, the event variable value is set to a positive and non-zero value. See Section 6.8.6.2 for an example of its use.

6.8.9 CALL DELETE Statement

The CALL DLETE (XVM/DOS form), CALL DELETE (XVM/RSX form) statement is used to remove the named file from the directory of the device connected to logical unit u and to return the storage space once used by that file to the pool of available free space.

6.8.9.1 CALL DLETE Statement in XVM/DOS - In XVM/DOS the format for CALL DLETE is:

```
CALL DLETE (u,a,v)
```

CALL DLETE must not be executed while there is a file open on logical unit u; otherwise, program execution will terminate and a system error message will appear on the console terminal. On completion of statement execution, if the file was found and deleted, the variable v is returned a value of 1. If not, v is set to zero.

The file name and extension are formed as described in Section 6.8.4.1 on CALL SEEK in XVM/DOS.

DOS
Example

```
DIMENSION A(2)
DATA A/5HT1ΔΔΔ,4HΔBIN/
CALL DLETE (10,A,J)
```

Here the CALL DLETE statement deletes a file named T1 BIN on logical unit 10.

6.8.9.2 CALL DELETE Statement in XVM/RSX - In XVM/RSX, the CALL DELETE statement is written:

```
          CALL DELETE (u,nHname,nHext[,ev])
```

u is a logical unit number (a positive integer constant or variable).

n is the number of characters (an unsigned decimal integer constant) in "name" and in "ext".

name is a string of from one to five ASCII characters representing the file name.

ext is a string of from one to three ASCII characters representing the file name extension.

ev is an optional integer event variable

If CALL DELETE is executed while there is a file open on logical unit u, or if the named file does not exist on that unit, the command is ignored and a negative value is stored in the event variable to indicate the source of the error. Otherwise, the file is deleted, and the event variable is set to a positive, non-zero variable to indicate successful execution.

RSX
Example

```
     CALL DELETE (19,2HT1,3HBIN)
```

In this example, file T1 BIN on logical unit 19 is deleted.

6.8.10 Additional I/O Subroutines

A few additional subroutines which can be used by XVM/DOS programs are described in the FORTRAN IV XVM Operating Environment Manual. For XVM/RSX, there are numerous FORTRAN-callable subroutines which are peculiar to each peripheral device. These are explained in the XVM/RSX System Manual.

6.9 LEGITIMATE INPUT/OUTPUT STATEMENT SEQUENCES

To this point, this chapter has described the individual input/output statements which exist under FORTRAN. The order in which these statements appear is important because only certain sequences are permitted.

6.9.1 Direct Access File I/O Sequences

To perform direct access I/O to a file on logical unit u, the file must be opened using the CALL DEFINE statement. CALL DEFINE cannot be executed if unit u is active with a previously opened file of any sort, direct access or not. Once the file is open, only READ and WRITE statements, in any sequence, are permitted. I/O may be formatted or unformatted, but not both in the same file. Once I/O is completed, the file must be closed using either the ENDFILE or the CALL CLOSE statement to declare logical unit u free for access to other files. Once the direct access file is closed, it is legitimate to issue CALL FSTAT, CALL RENAME and CALL DELETE statements, in any order, to that logical unit.

6.9.2 Sequential Access Named-File Input Sequences

Before data can be read from a named sequential access file, the file must be opened with the CALL SEEK statement. CALL SEEK cannot be executed on a logical unit which is active with a previously opened file, whether sequential access or not. Once the file is open, only READ statements are permitted. Data may be either formatted or unformatted but not both in the same file. The END=s option in the READ statement can be used to detect end-of-file or end-of-medium when encountered. After all READ statements have been issued, the file must be closed using either the ENDFILE or the CALL CLOSE statement. This will declare the logical unit free for access to other files. Again, when the logical unit is free, it is permitted to execute CALL FSTAT, CALL RENAME and CALL DELETE statements, in any order, to the logical unit.

6.9.3 Sequential Access Named-File Output Sequences

If a logical unit is free (no file open) a CALL ENTER statement may be executed to open a named-file for sequential output. If a file of the given name already exists, it will be replaced with the new file once it is closed. Following CALL ENTER, only a sequence of WRITE statements is allowed. Data may be either formatted or unformatted but not both in the same file. After all WRITEs have been executed, the file must be closed and the logical unit freed by use of the ENDFILE or CALL CLOSE statement. As before, once the logical unit is free, CALL FSTAT, CALL RENAME and CALL DELETE are allowed in any order, to that unit.

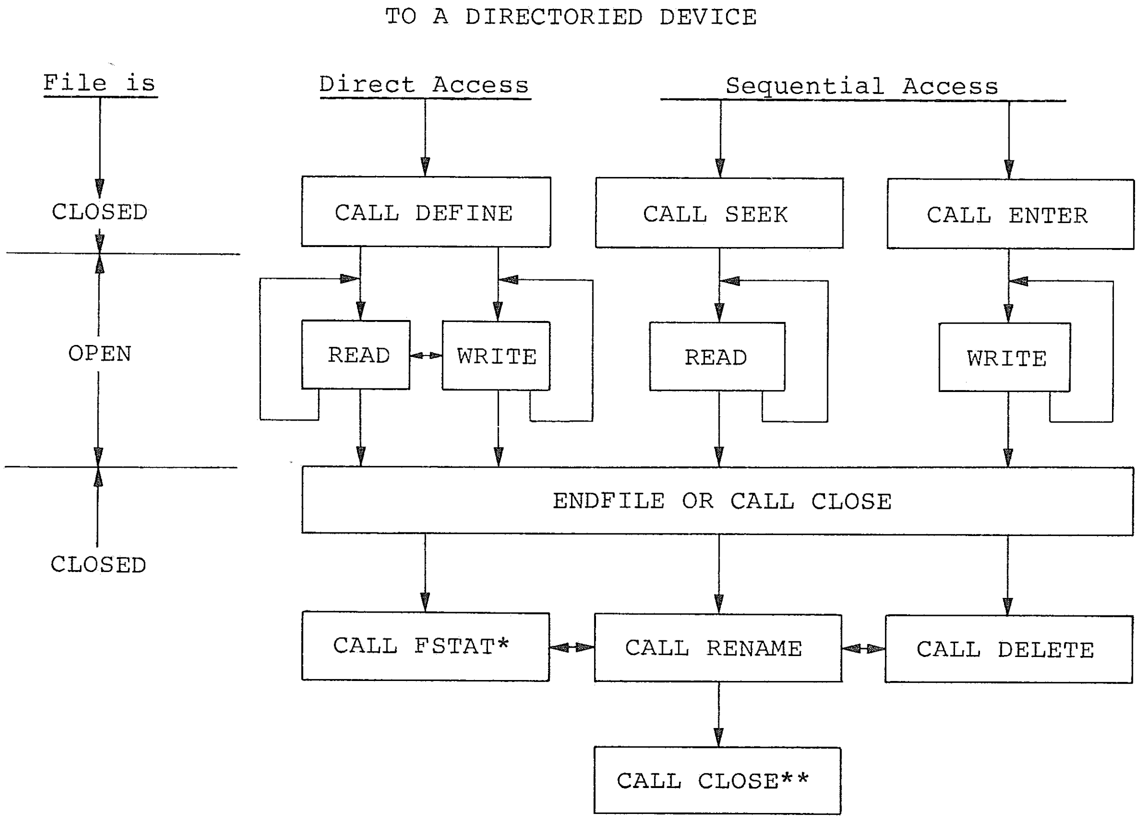
6.9.4 Sequential Access Unnamed File I/O Sequences

If I/O to disk, magtape or DECtape is to be performed without explicit reference to a named file, there are two modes of operation: one in which the device has a directory and I/O is performed on a file whose name is assigned automatically by the system or one in which the device has no directory and data is recorded in sequence from a physical starting point. In the XVM/RSX system, sequential I/O to disk and DECtape must be performed with reference to a named file and I/O to magtape cannot. The only situation discussed in this section which is relevant to XVM/RSX is unnamed file I/O to non-directoried magtape.

In the first case, which applies only in XVM/DOS systems, execution of a sequential READ or WRITE statement to the directoried versions of the device handlers without a prior CALL SEEK, CALL ENTER or (in the case of DECtape) a REWIND statement, causes the system to open a file of the name .TM0uu OTS, where uu is the logical unit number represented in decimal ASCII form. When the file is first being written, it is typical either to use multiple WRITE statements followed by an ENDFILE to close the file or sequences of WRITE, BACKSPACE, READ to verify the written data as the file is being created. The file may also be closed by execution of a REWIND command, which executes an implied ENDFILE. Once the newly created file is closed, its name is recorded in the device's file directory. If a file of the same name already appeared in the directory, the old version is deleted once the new one appears. If after the file is closed it is reopened by issuing a READ statement, processing restarts at the beginning of the file. Until the file is closed with an ENDFILE, or REWIND statement, only READ and BACKSPACE statements are legitimate. After the file is closed, it is permitted to execute CALL FSTAT, CALL RENAME or CALL DELETE. Since the file .TM0uu OTS is not automatically deleted on completion of the user program, use of CALL DELETE will be necessary to delete the temporary file to free up storage on the device for the next user.

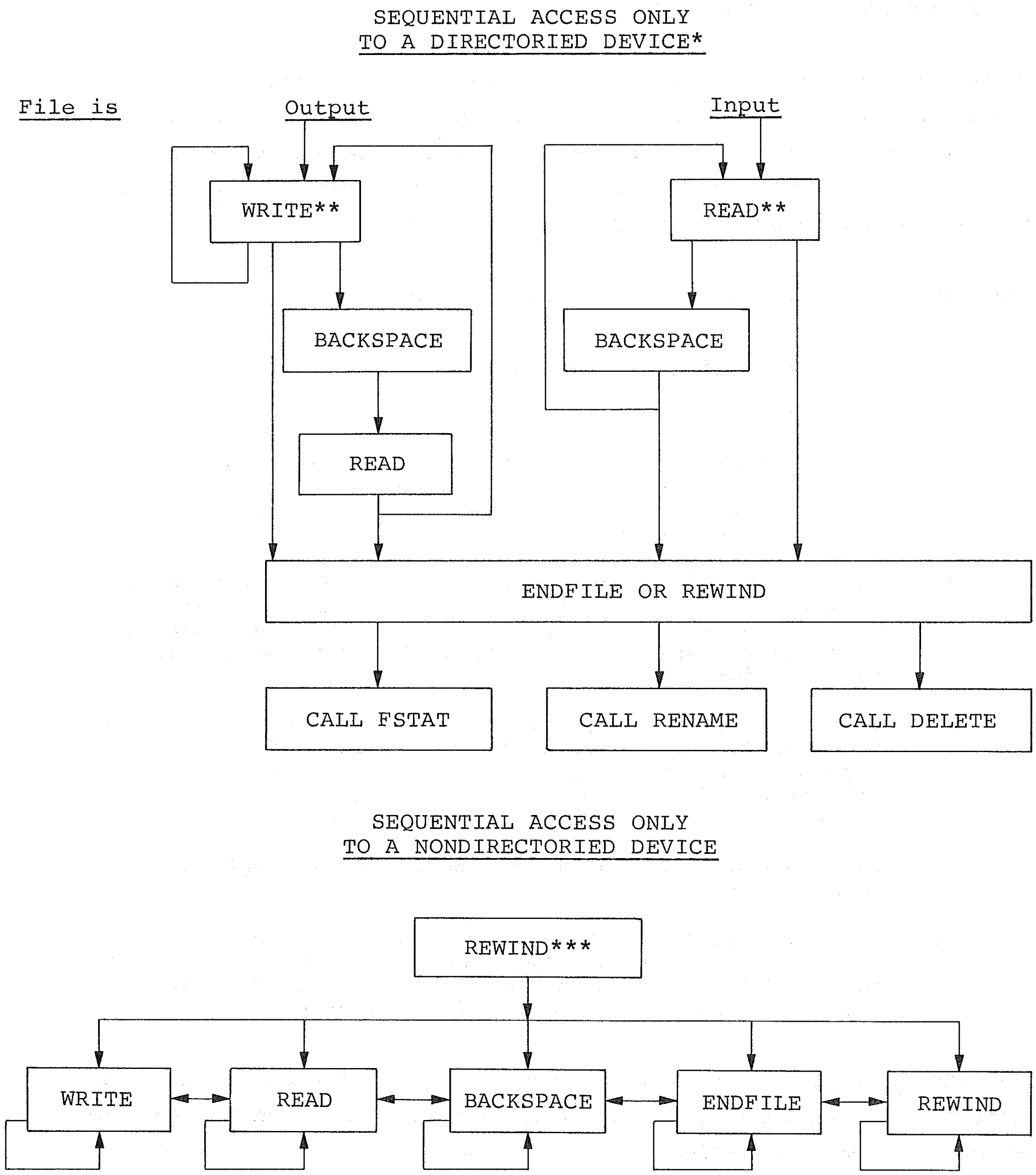
In the second case, which does not apply to disk (nor to DECtape in XVM/RSX), the device has no directory. For DECtape it is necessary to start this mode of I/O with a REWIND command, but to magtape this is optional. Thereafter, any combination of WRITE, BACKSPACE, READ, ENDFILE and REWIND commands is legal. Several files can be written in sequence by executing multiple WRITE statements, an ENDFILE statement, more WRITE statements, and so on.

In all cases, data may be either formatted or unformatted but not both in the same file.



*CALL FSTAT is used only in XVM/DOS.
**CALL CLOSE after CALL RENAME is used only in XVM/RSX.

Figure 6-1
Named File Input/Output Sequences



SEQUENTIAL ACCESS ONLY
TO A DIRECTORIED DEVICE*

SEQUENTIAL ACCESS ONLY
TO A NONDIRECTORIED DEVICE

*Applies only to XVM/DOS, not XVM/RSX.
**Assumes a file named .TMØuu OTS.
***Required on DECtape only in XVM/DOS.

Figure 6-2
Unnamed File Input/Output Sequences

6.10 ENCODE AND DECODE STATEMENTS

These two statements perform data transfers according to format specifications, whether referenced explicitly or implied, translating data from internal format to alphanumeric (ASCII) format, or vice versa. Unlike conventional formatted I/O statements, however, these data transfers take place entirely within memory.

The ENCODE and DECODE statements are written as follows:

```
ENCODE(c,a,[f][,ERR=s])[list]

DECODE(c,a,[f][,ERR=s])[list]
```

c is an integer constant or variable representing the number of ASCII characters that are to be converted or that are to result from the conversion. (This is analogous to the length of an external record.)

a is the name of a real, double integer or double precision array. In the ENCODE statement, this array receives the encoded ASCII characters. In the DECODE statement, it contains the ASCII characters that are to be translated to internal format.

f is the statement label of a FORMAT statement or the name of an array that contains format specifications. Only one record can be transmitted, that is, the occurrence of a "/" (slash) format specification separator or of format reversion will cause an error condition.

s is an executable statement label. The ERR option is described in Section 7.2.

list is an I/O list. In the ENCODE statement, the I/O list contains the data that is to be converted to ASCII format. In the DECODE statement, the list receives the data that has been translated from ASCII to internal format.

The ENCODE statement is analogous to a WRITE statement in the sense that the I/O list contains the data that is to be transmitted. The ENCODE statement converts that data from the form specified by the FORMAT statement, format array or implied format to ASCII format and stores those ASCII characters in array a. The data is taken from the elements in the I/O list from left to right, converted, and stored in the array in the order of subscript progression. If more variables are listed than can be stored in the array, the excess variables are ignored. If not enough variables are specified to fill the array, the remainder of the array is filled with blanks (spaces).

The DECODE statement can be likened to a READ statement, because the internal-format data resulting from the execution of the statement is assigned to the elements in the I/O list. The DECODE statement takes the ASCII characters from array a, processing the array in the order of subscript progression, converts that data to the form specified by the FORMAT statement, format array, or implied format, and assigns the data to the elements of the I/O list from left to right.

Conversion is performed according to a FORMAT statement or format array or, in their absence, according to the standard implied format rules specified in Section 7.7. In the case where the data-directed I/O form is used, the data-directed output form 'VAR' = value is stored as follows

```
        I = 5
        J = 1
        ENCODE (c,ARRAY,,ERR=100)I,J
```

where

$$c=3+\text{number of characters in name}+\begin{cases}1 & \text{if logical}\\ 7 & \text{if integer}\\ 12 & \text{if double integer}\\ 16 & \text{if real}\\ 20 & \text{if double precision}\end{cases}$$

NOTE

> Forms control characters are not analyzed on ENCODE and are stored explicitly if they appear in the format specification.

The number of ASCII characters that can be handled by the ENCODE or DECODE statement is dependent on the data type of the array b in that statement. An INTEGER*2 array, for example, can contain up to two characters per element, so the maximum number of characters is twice the number of elements in that array.

The interaction between format control and the I/O list is the same as for a formatted I/O statement.

Example

```
        DIMENSION A(3),K(3)
        DATA A /'1234','5678',9012'/
        DECODE (12,100,A) K
    100 FORMAT (3I4)
```

Execution of the DECODE statement causes the 12 ASCII characters in array A to be converted to Integer format (specified by statement 100) and stored in array K, as follows:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

CHAPTER 7

FORMAT STATEMENTS

7.1 OVERVIEW

FORMAT statements are nonexecutable statements used in conjunction with formatted I/O statements and with ENCODE and DECODE statements. The FORMAT statement describes the format in which data fields are transmitted, and the data conversion and editing to be performed to achieve that format. In lieu of an explicit reference to a FORMAT statement, a format stored in an array (see Section 7.6) or a system-supplied implied format (see Section 7.7) may be used.

The FORMAT statement is written:

FORMAT $(d_1 k_1 d_2 k_2 \ldots d_n k_n d_{n+1})$

d_i represents a group of record separators (one or more slash (/) characters), which must be present if argument r_i is present. There are two exceptions: d_1 and d_{n+1} are entirely optional. The effect of record separators is described in Section 7.4.

k_i represents a record format list which describes the format of all data fields in record i.

The general form for a record format list is:

$F_1, F_2, \ldots, F_n$

F_i represents either a simple field descriptor, f, or a group of field descriptors enclosed in parentheses and preceded by a repeat count: $r(f_1, f_2, \ldots, f_n)$.

A field descriptor in a format specification appears in one of three forms:

[r]C or [r]Cw or [r]Cw.d

Where C is a format code which must always be given; w (when present) specifies the field width; and d (when present) specifies the number of characters in the field that appear to the right of the decimal point. The term r represents an optional repeat count, which specifies that the field descriptor is to be applied to r successive fields. If the repeat count is omitted, it is presumed to be 1. For a discussion of individual and group field repetition, see Section 7.2.15.

The entire list of field descriptors, field and record separators and the enclosing parentheses is called the format specification.

The terms r, w, and d must all be unsigned integer constants less than or equal to 255.

The most commonly used field separator is a comma. A slash (/) may also be used; it has the additional function of being a record terminator. The functions of the field separators are described in detail in Section 7.4.

The field descriptors used in format specifications are as follows:

1.	Integer or Double Integer (both Decimal and Octal):	Iw, Ow
2.	Logical:	Lw
3.	Real or Double Precision:	Fw.d, Ew.d, Dw.d, Gw.d
4.	Literal and editing:	Aw, Rw, nH, '...', nX, Tn

(In the alphanumeric and editing field descriptors, n specifies a number of characters or character positions.)

Any of the F, E, D, or G field descriptors may be preceded by a scale factor of the form:

[±] nP

where n is an optionally signed integer constant that specifies the number of positions the decimal point is to be scaled to the left or right. The scale factor is described in Section 7.2.14.

During data transmission, the object program scans the format specification from left to right. Data conversion is performed by correlating the data values in the I/O list with the corresponding field descriptors. In the case of H field descriptors and alphanumeric literals, data transmission takes place entirely between the field descriptor and the external record. The interaction between the format specification and the I/O list is described in detail in Section 7.8.

Data written out to a storage device under format control can be read in again using the same format. Also, within limits, one type of device may be used in lieu of another without requiring a change in the user's program. This is referred to as device independence. Note, however, that a FORMAT statement is treated differently for printing devices than it is for non-printing devices. For printing devices, the FORTRAN Object Time System interprets the first character of each record as a carriage control indicator. This is discussed in Section 7.3.

7.2 FIELD DESCRIPTORS

The individual field descriptors that may appear in a format specification are described in detail in the following sections. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeros.

7.2.1 Field Width Constraints

All field descriptors which specify data conversion (A, D, E, F, G, I, L, R) indicate a field width, w. The field width must be large enough to contain all characters, including decimal point and sign, required to constitute the data value. If on output the field width is too small to contain all the converted characters, the field is filled with asterisks instead, including the decimal point if pertinent; e.g., **.***. If on input the external data field is wider than the width indicated in the corresponding field descriptor, only the leftmost (most significant) characters are retained; once the first w characters have been input, the remaining characters in the external field are lost. If on input the external data field is shorter than the width specified in the corresponding field descriptor, trailing blanks are assumed.

7.2.2 I Field Descriptor

The I field descriptor governs the translation of both integer and double integer data. It appears as:

Iw or rIw

The I field descriptor causes an input statement to read w characters from the external record, convert them to an internal format, and store them in the associated integer element of the I/O list. The external data must be an integer or double integer; it must not contain a decimal point or exponent field. The I field descriptor interprets an all-blank field as a zero value. If the value of the external field exceeds the range of the corresponding integer list element, an error occurs. If the first non-blank character of the external field is a minus symbol, the I field descriptor causes the field to be stored as a negative value; a field preceded by a plus symbol, or an unsigned field, is treated as a positive value. For example:

Format	External Field	Internal Representation
I4	2788	2788
I3	-26	-26
I9	ΔΔΔΔΔΔ312	312
I9	3.12	not permitted; error
I3	-Δ5	-05
I11	+1234567890	1234567890

The I field descriptor causes an output statement to transmit the value of the associated integer I/O list element to an external field w characters in length, right justified, replacing any leading zeros with spaces. If the value of the list element is negative, the field will have a minus symbol as its leftmost non-blank character. Space must therefore be included in w for a minus symbol if one is expected. Plus symbols, on the other hand, are suppressed and need not be accounted for in w. If w is too small to contain the output value, the entire external 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	29.812	not permitted; error

Example 1

```
      READ (3,100) IHOUR, IMIN, ISEC
100   FORMAT (I2,1X,I2,1X,I2)
```

In this first example, three integer fields of width 2, separated from each other by a space, are input from logical unit 3. This might be used at a terminal to specify hours, minutes and seconds. Consider the following input data and the values stored in the variables specified in the I/O list.

input stream	IHOUR	IMIN	ISEC
03Δ15Δ20	3	15	20
19:35:00	19	35	0
3Δ15Δ20	30	50	0
3:15:20		error	

The input streams in lines 1 and 2 are correct because each integer is comprised of two digits and each integer field is separated by one character (which does not have to be a space). Line 3, though improperly formed, does not violate the rules. An embedded blank is treated as a zero, hence 3Δ is converted to 30 for the first value. The 1 is discarded (skipped) in accord with the 1x descriptor and so 5Δ is taken as the second field and converted to 50. The 2 is ignored because of the second 1X descriptor and finally the last argument value is simply 0.

Example 2

```
      I1=333
      I2=65000
      I3=10
      WRITE (6,20) I1,I2,I3
20    FORMAT (1X,I3,1X,I3,1X,I3)
```

In this example, assuming that the output is going to a printing device, such as a line printer, the first character in the FORMAT statement is treated as a carriage control indicator (see Section 7.3) and not as a space character. The first and third variable values can be expressed in three digits and so are output as expected.

```
333Δ***ΔΔ10
```

Because the value of the second variable cannot be expressed in a field width of 3, asterisks are printed instead.

7.2.3 O Field Descriptor

The O field descriptor governs the transmission of octal integer or octal double integer values. It appears as follows:

```
Ow   or   rOw
```

The O field descriptor causes an input statement to read w characters from the external record, assigning them to the associated I/O list element as an octal value. The list element must be of integer, double integer or logical type. The external field must contain only the numerals 0 through 7; it must not contain a sign, a decimal point, an exponent field, or (for octal double integers) a D prefix. For example:

Format	External Field	Internal Octal Representation
O5	32767	32767
O4	16234	1623
O6	13ΔΔΔΔ	130000
O3	97Δ	not permitted; error
O6	-1	777777
O12	400000000000	400000000000

The O field descriptor causes an output statement to transmit the value of the associated I/O list element, right justified, to a field "w" characters long. If the data does not fill the field, leading zeros are inserted; if the data exceeds the field width, only the rightmost "w" characters are transmitted. No signs are output; a negative value is transmitted in its octal (two's complement) form. The I/O list element must be of integer, double integer or logical type. For example:

Format	Internal (octal) Value	External Representation
O6	77777	077777
O2	14261	61
O4	33	0033
O5	13.52	not permitted; error
O12	123456654321	123456654321

7.2.4 F Field Descriptor

The F field descriptor specifies the data conversion and editing of real or double precision values. It is written as shown below. (In all appearances of the F field descriptor, w must be greater than or equal to d+1.)

```
Fw.d  or  rFw.d
```

The corresponding I/O list element must be of real or double precision type.

On input, the F field descriptor causes w characters to be read from the external record and to be assigned as a real value to the related I/O list element. If the first non-blank character of the external field is a minus sign, the field is treated as a negative value; a field that is preceded by a plus sign, or an unsigned field, is considered to be positive. An all-blank field is considered to have a value of zero.

If the external field contains neither a decimal point nor an exponent, it is treated as a real number of w digits, in which the rightmost d digits are to the right of the decimal point. If the field contains an explicit decimal point, the location of that decimal point overrides the location specified by the field descriptor. If the field contains an exponent (in the same form as described in section 2.2.2 for real constants or 2.2.3 for double precision constants), that exponent is used in establishing the magnitude of the value before it is assigned to the list element. The number itself must be restricted to 11 digits to ensure proper conversion; otherwise, the results would be unpredictable. For example, here are some of the more commonly used input fields:

Format	External Field	Internal Representation
F6.3	Δ13457	13.457
F6.3	1.3457	1.3457
F9.2	-21367.	-21367.

On output, the F field descriptor converts the value of the related I/O list element to a minus sign (if negative), an integer portion, a decimal point, and a fractional part rounded to d significant digits. If the converted data consists of fewer than w characters, leading spaces are inserted so that the number is right-justified; if the data exceeds w characters, the entire field is filled with asterisks.

The total field width specified must be large enough to accommodate a minus sign, if one is expected (plus signs are suppressed), at least one digit to the left of the decimal point which may be zero, the decimal point itself, and d digits to the right of the decimal. For this reason, w should always be greater than or equal to (d+3). Examples follow:

Format	Internal Value	External Representation
F8.5	2.3547188	Δ2.35472
F9.3	8789.7361	Δ8789.736
F2.3	51.44	not permitted; error
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20
F3.2	-.2	***

Note in the third example that the format field descriptor is incorrect because the field width, w, is less than d.

7.2.5 E Field Descriptor

The E field descriptor handles the transmission of real data in exponential format. It appears as follows:

Ew.d or rEw.d

The corresponding I/O list element must be of real or double precision type.

The E field descriptor causes a READ statement to input w characters from the external record. It interprets and assigns that data in exactly the same way as the F field descriptor. The number itself must be restricted to 11 digits to ensure proper conversion; otherwise, the results would be unpredictable. For example:

Format	External Field	Internal Representation
E9.3	734.432E3	734432.0
E12.4	ΔΔ1022.43E-6	1022.43E-6
E15.3	52.3759663ΔΔΔΔΔ	52.3759663
E12.5	210.5271D+10	210.5271E10

Note that in the last example the E field descriptor disregards the double precision connotation of a D exponent field indicator and treats it as though it were an E indicator.

The E field descriptor causes a WRITE statement to transmit the value of the corresponding list element to an external field w characters in width, right justified. If the number of characters in the converted data is less than w, leading spaces are inserted; if the number of characters exceeds w, the entire field is filled with asterisks.

Data output under control of the E field descriptor is transmitted in a standard form, consisting of a minus sign if the value is negative (plus signs are suppressed), a zero, a decimal point, d digits to the right of the decimal, and a 4-character exponent of the form:

E±nn

where nn is a 2-digit integer constant. The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

Because w must be large enough to include a minus sign (if one is expected), a zero, a decimal point, and an exponent, in addition to d digits, w should always be equal to or greater than (d+7). Some examples are:

Format	Internal Value	External Representation
E9.2	475867.222	Δ0.48E+06
E12.5	475867.222	Δ0.47587E+06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****

7.2.6 D Field Descriptor

The D field descriptor governs the transmission of real or double precision data. It appears as follows:

Dw.d or rDw.d

On input, the D field descriptor functions exactly as an equivalent E field descriptor, except that the input data is converted and assigned as a double precision entity, as in the following examples:

Format	External Field	Internal Representation
D10.2	12345ΔΔΔΔΔ	12345000.0D0
D10.2	ΔΔ123.45ΔΔ	123.45D0
D15.3	367.4981763D+04	3.674981763D+06
D13.6	Δ+34567890123	34567.890123

At most 11 significant digits can be input without truncation. The largest positive number is 34359738367. Larger values are converted to meaningless values.

On output the effect of the D field descriptor is identical to that of the E field descriptor, except that the D exponent field indicator is used in place of the E indicator. For example:

Format	Internal Value	External Representation
D14.3	0.0363	ΔΔΔΔΔ0.363D-01
D23.12	5413.87625793	ΔΔΔΔΔ0.541387625793D+04
D9.6	1.2	*********

7.2.7 G Field Descriptor

The G field descriptor transmits real or double precision data in a form that is in effect a combination of the F and E field descriptors. It appears as follows:

Gw.d or rGw.d

On input, the G field descriptor functions identically to the F field descriptor (see Section 7.2.3).

On output, the G field descriptor causes the value of the associated I/O list element to be transmitted to an external field w characters in length, right justified. The form in which the data is output is a function of the magnitude of the data itself, as described in Table 7-1.

Table 7-1
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
$m < 0.1$	Ew.d
$0.1 \leq m < 1.0$	F(w-4).d, 4X
$1.0 \leq m < 10.0$	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).0, 4X
$m \geq 10^{d}$	Ew.d

The 4X field descriptor, which is (in effect) inserted by the G field descriptor for values within its range, specifies that four spaces are to follow the numeric data representation. The X field descriptor is described in Section 7.2.12.

The field width, w, must include space for a minus sign, if one is expected (plus signs are suppressed), at least one digit to the left of the decimal point, the decimal point itself, d digits to the right of the decimal, and (for values that are outside the effective range of the G field descriptor) a 4-character exponent. Therefore, w should always be equal to or greater than (d+7). Examples of G output conversions are:

Format	Internal Value	External Representation
G13.6	0.01234567	Δ0.123457E-01
G13.6	-0.12345678	-0.123457ΔΔΔΔ
G13.6	1.23456789	ΔΔ1.23457ΔΔΔΔ
G13.6	12.34567890	ΔΔ12.3457ΔΔΔΔ
G13.6	123.45678901	ΔΔ123.457ΔΔΔΔ
G13.6	-1234.5678901	Δ-1234.57ΔΔΔΔ
G13.6	12345.678901	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901	ΔΔ123457.ΔΔΔΔ
G13.6	-1234567.8901	-0.123457E+07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	ΔΔΔΔΔ0.012346
F13.6	-0.12345678	ΔΔΔΔ-0.123457
F13.6	1.23456789	ΔΔΔΔΔ1.234568
F13.6	12.34567890	ΔΔΔΔ12.345679
F13.6	123.45678901	ΔΔΔ123.856789
F13.6	-1234.5678901	Δ-1234.567890
F13.6	12345.678901	Δ12345.678901
F13.6	123456.78901	123456.789010
F13.6	-1234567.8901	****** ******

7.2.8 L Field Descriptor

The L field descriptor governs the transmission of logical data. It appears as:

Lw or rLw

The corresponding I/O list element must be of logical or integer type.

The L field descriptor causes an input statement to read w characters from the external record. If the first non-blank character of that field is the letter T, the value .TRUE. is assigned to the associated I/O list element. If the first non-blank character of the field is the letter F or any other character but T, or if the entire field is blank, the value .FALSE. is assigned.

The L field descriptor causes an output statement to transmit either the letter T, if the value of the related list element is .TRUE., or the letter F, if the value is .FALSE., to an external field w characters wide. Any non-zero value in this situation is treated as .TRUE.. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces. For example:

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

7.2.9 A Field Descriptor

The A field descriptor controls the transmission of alphanumeric data. It is written as shown below.

Aw or rAw

On input, the A field descriptor causes w characters to be read from the external record and stored in ASCII format in the associated I/O list element. The maximum number of characters that can be stored in a variable or array element depends on the data type of that element, as follows:

I/O List Element	Maximum Number of Characters
Logical	2
Integer	2
Double Integer	5
Real	5
Double Precision	5

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost two or five characters (depending on the data type of the variable or array element) are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that can be stored, w characters are assigned to the list element, left justified, and trailing spaces are added to fill the variable or array element. For example:

Format	External Field	Internal Representation
A1	HELLO!	HΔ (Integer)
A2	HELLO!	HE (Integer)
A3	HELLO!	HELΔΔ (Real)
A4	HELLO!	HELLΔ (Real)
A5	HELLO!	HELLO (Real)
A6	HELLO!	ELLO! (Real)

On output, the A field descriptor causes the contents of the related I/O list element to be transmitted to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right-justified with leading spaces. If the list element contains more than w characters, only the leftmost w characters are transmitted. For example:

Format	Internal Value	External Representation
A5	OHMSΔ	OHMSΔ
A5	VOLTS	VOLTS
A2	VOLTS	VO
A6	VOLTS	ΔVOLTS

The A and R field descriptors perform similar conversions, the latter storing characters internally right-justified rather than left-justified. Because of the manner in which characters are stored in integer or logical (single word) variables, characters stored in A format are not in the same positions in the word as characters stored in R format. Thus, it is not possible to output characters from an integer or logical variable (or array element) using A format that were input using R format, or vice versa.

7.2.10 R Field Descriptor

The R field descriptor controls the transmission of alphanumeric data. It is written as follows:

Rw or rRw

On input, the R field descriptor causes w characters to be read from the external record and stored in ASCII format in the associated I/O list element. The maximum number of characters that can be stored in a variable or array element depends on the data type of that element, as shown in Section 7.2.9.

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost two or five characters (depending on the data type of the variable or array element) are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that can be stored, w characters are stored, right justified, and leading spaces are added to fill the variable or array element. For example:

Format	External Field	Internal Representation
R1	HELLO!	Δ! (Integer)
R2	HELLO!	O! (Integer)
R3	HELLO!	ΔΔLO! (Real)
R4	HELLO!	ΔLLO! (Real)
R5	HELLO!	ELLO! (Real)
R6	HELLO!	ELLO! (Real)

On output, the R field descriptor causes the contents of the related I/O list element to be transmitted to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right-justified with leading spaces. If the list element contains more than w characters, only the rightmost w characters are transmitted. For example:

Format	Internal Value	External Representation
R5	OHMSΔ	OHMSΔ
R5	VOLTS	VOLTS
R2	VOLTS	TS
R6	VOLTS	ΔVOLTS

At the end of Section 7.2.9, a restriction pertaining to A and R format usage is discussed.

7.2.11 H Field Descriptor

The H field descriptor takes the form of a Hollerith constant:

$$nHc_1c_2c_3 \ldots c_n$$

n specifies the number of characters that are to be transmitted

c is an ASCII character.

When the H field descriptor appears in a format specification, data transmission takes place between the external record and the field descriptor itself.

The H field descriptor causes an input statement to read n characters from the external record and to place them in the field descriptor, with the first character appearing immediately after the letter H. Any characters that had been in the field descriptor prior to input are replaced by the input characters.

The H field descriptor causes an output statement to transmit the n characters in the field descriptor following the letter H to the external record in ASCII form. An example of the use of H field descriptors for input and output follows:

```
     WRITE (1,100)
100  FORMAT (41HΔENTERΔPROGRAMΔTITLE,ΔUPΔTOΔ20ΔCHARACTERS)
     READ (1,200)
200  FORMAT (20HΔΔTITLEΔGOESΔHEREΔΔΔ)
```

The WRITE statement transmits the characters from the H field descriptor in statement 100 to logical unit 1, assumed to be connected to the user's terminal. The READ statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the words TITLE GOES HERE and the two leading and three trailing spaces; if the user enters fewer than 20 characters, the remainder of the H field descriptor is filled with spaces to the right.

7.2.11.1 Alphanumeric Literals - An alphanumeric literal (an ASCII character string enclosed in apostrophes) can be used in place of an H field descriptor. Both types of format specifiers function identically.

The apostrophe character is written within an alphanumeric literal as two apostrophes. For example:

```
50  FORMAT ('TODAY''SΔDATEΔIS:Δ',I2,'/',I2,'/',I2)
```

A pair of apostrophes used in this manner is considered to be a single character.

For historical reasons, the characters double quote and dollar sign can be used rather than single quote to enclose an alphanumeric literal, but single quote is the preferred character for compatibility with other DEC FORTRANs. 'ABC' and "ABC" and ABC are equivalent.

Example

Hollerith field descriptors may be combined with other field descriptors. For instance, the following output to a printing device gives a readable answer to a numerical computation.

```
          I=2
          ICUBED=I**3
          WRITE (1,10)I,ICUBED
      10  FORMAT(1X,I6, 4H**3=,I6)
```

The resulting printout would be:

```
ΔΔΔΔΔ2**3=ΔΔΔΔΔ8
```

Note that the comma following a Hollerith field (in this case following the =) is optional, but it is generally retained for readability to help separate field descriptors.

7.2.12 X Field Descriptor

The X field descriptor is written:

```
nX
```

The X field descriptor causes an input statement to skip over the next n characters in the input record.

The X field descriptor causes an output statement to transmit n spaces to the external record. For example:

```
           WRITE (5,90) NPAGE
      90   FORMAT (13H1PAGEΔNUMBERΔ,I2,16X,23HGRAPHICΔANALYSIS,ΔCONT.)
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn                  GRAPHIC ANALYSIS, CONT.
```

where "nn" is the current value of the variable NPAGE. The numeral 1 in the first H field descriptor is not printed, but is used to advance the printer paper to the top of a new page. Printer carriage control is explained in Section 7.3.

7.2.13 T Field Descriptor

The T field descriptor, which appears as follows:

```
Tn
```

is a tabulation specifier. The value of n on output must be greater than or equal to two, but not greater than the number of characters allowed in the external record. On input, the value of n must be greater than or equal to one.

On input, the T field descriptor causes the external record to be positioned to its nth character position. For example, if a READ statement input a record containing:

```
ABCΔΔΔXYZ
```

under control of the FORMAT statement:

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

the READ statement would input the characters XYZ first, then the characters ABC.

On output to non-printing devices, the T field descriptor states that subsequent data transfer is to begin at the nth character position of the external record. For output to a printing device, data transfer begins at position (n-1). The first position of a printed record is reserved for a carriage control character (see Section 7.3) which is never printed.

Examples:

```
    WRITE (3,25)
25  FORMAT (T51,'COLUMNΔ2',T21,'COLUMNΔ1')
```

would cause the following line to be printed:

```
Position 20                    Position 50
     ↓                              ↓
     COLUMN 1                       COLUMN 2
```

In the next example,

```
        READ (2,10)
     10 FORMAT (T4,'ABC')
```

if the input string is ABCDEFGHI, then the FORMAT statement is changed to

```
     10 FORMAT (T4,'DEF')
```

In this last example,

```
        WRITE (6,20)
     20 FORMAT (1X,'IΔWON''T',T4,'/////ΔWILL')
```

the printout beginning in column 1 will appear as:

```
     IΔW̸Ø̸N̸/T̸ΔWILL
```

7.2.14 Scale Factor

The location of the decimal point in real and double precision values can be altered during input or output through the use of a scale factor, which takes the form:

nP

where n is a signed or unsigned decimal integer constant in the range -127 to +127 specifying the number of positions the decimal point is to be moved to the right or left.

A scale factor may appear anywhere in a format specification, but must precede the field descriptors with which it is to be associated. It is normally written as follows:

```
     nPFw.d       nPEw.d       nPDw.d       nPGw.d
```

Data input under control of one of the above field descriptors is multiplied by 10^{-n} before it is assigned to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left; a -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. If the external field contains an explicit exponent, however, the scale factor has no effect. For example:

Format	External Field	Internal Representation
3PE10.5	ΔΔΔ37.614Δ	.037614
3PE10.5	ΔΔ37.614E2	3761.4
-3PE10.5	ΔΔΔΔ37.614	37614.0

The effect of the scale factor on output depends on the type of field descriptor with which it is associated. For the F field descriptor, the value of the I/O list element is multiplied by 10^n before being transmitted to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

Values output under control of an E or D field descriptor with scale factor are adjusted by multiplying the basic real constant portion of each value by 10^n and subtracting n from the exponent. Thus a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

The effect of the scale factor is suspended while the magnitude of the data to be output is within the effective range of the G field descriptor, since it supplies its own scaling function. The G field descriptor functions as an E field descriptor when the magnitude of the data value is outside its range; the effect of the scale factor is therefore the same as described for that field descriptor.

Note that on input, and on output under control of an F field descriptor, a scale factor actually alters the magnitude of the data; on output, a scale factor attached to an E, D, or G field descriptor merely alters the form in which the data is transmitted. Note also that on input a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right, while on output the effect is just the reverse.

If no scale factor is attached to a field descriptor, a scale factor of zero is assumed. Once a scale factor has been specified, however, it applies to all subsequent real and double precision field descriptors in the same format specification, unless another scale factor appears; that scale factor then assumes control. A scale factor of zero can only be reinstated by an explicit 0P specification.

Some examples of scale factor effect on output are:

Format	Internal Value	External Representation
3PE12.3	-270.139	-270.139E+00
1PE12.3	-270.139	ΔΔ-2.701E+02
1PE12.2	-270.139	ΔΔΔ-2.70E+02
-1PE12.2	-270.139	ΔΔΔ-0.03E+04

7.2.15 Grouping and Group Repeat Specifications

Any field descriptor (except H, T or X) may be applied to a number of successive data fields by preceding that field descriptor with an unsigned integer constant, called a repeat count, that specifies the number of repetitions. For example, the statements:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
```

and

```
20  FORMAT (3E12.4,4I5)
```

have the same effect.

Similarly, a group of field descriptors may be repeatedly applied to data fields by enclosing those field descriptors in parentheses, with an unsigned integer constant, called a group repeat count, preceding the opening left parenthesis. For example:

```
50  FORMAT (2I8,3(F8.3,E15.7))
```

is equivalent to:

$$\texttt{50 FORMAT (I8,I8,}\underbrace{\texttt{F8.3,E15.7}}_{1}\texttt{,}\underbrace{\texttt{F8.3,E15.7}}_{2}\texttt{,}\underbrace{\texttt{F8.3,E15.7}}_{3}\texttt{)}$$

An H or X field descriptor, which could not otherwise be repeated, may be enclosed in parentheses and treated as a group repeat specification, thus allowing it to be repeated a desired number of times.

If a group repeat count is omitted, it is presumed to be 1.

7.3 CARRIAGE CONTROL

The first character of every record transmitted to a printing device (line printer, Teletypewriter, VP15A Storage Scope and XY11 Plotter) is never printed; instead, it is interpreted as a carriage control character. The FORTRAN I/O system recognizes certain characters for this purpose; the effects of those characters are shown in Table 7-2.

Table 7-2
Carriage Control Characters

Character	Effect
Δ space	Advances one line (line feed)
0 zero	Advances two lines (double space)
1 one	Advances to top of next page (form feed)
+ plus	Does not advance (allows overprinting)
All others	Advances one line (line feed)

Note in Table 7-2 that any character other than those explicitly described is treated as though it were a space, and is deleted from the print line. Whatever action is specified by the carriage control character occurs prior to the printing of the record.

Records can be written to and read back from non-printing devices (disk, magtape, DECtape, and paper tape) using the FORMAT statements. It is permissible to omit the carriage control character at the head of each format record. However, data files created in this manner cannot later be transferred from, say, disk to line printer without loss of data because of the missing carriage control characters unless the data were read back under the original format and printed using a different format.

It is possible to write a program in which the source of input data (the type of device being used) as well as the destination of output data can be varied from one run to the next without necessitating rebuilding the program. This is called device independence. For example, the following statements

```
          READ (1,100) I,J,K
    100   FORMAT (3I6)
```

could be used to input data values from cards, paper tape, keyboard or a DECtape, magtape or disk file. Similarly, the following statements

```
          WRITE(6,200)I,J,K
    200   FORMAT (1X,3I6)
```

could be used to write out the values of three integer variables to either a line printer, teleprinter, paper tape, VP15A storage scope, or to a DECtape, magtape or disk file. When the data is written to a non-printing device, such as disk, the carriage control characters are left as is and not converted into the appropriate vertical format characters (e.g., line feed and form feed). Such a file can later be listed on a printing device using the PIP XVM program in XVM/DOS and a special command option (V) to perform the carriage control conversion.

Note that in the previous READ and WRITE examples that data written out under FORMAT statement 200 cannot be correctly retrieved by reading with FORMAT statement 100. However, the data could be reread using the identical FORMAT under which data was written.

Example 1

```
          READ(1,20)I,J
     20   FORMAT (2I2)
```

Input String	I Value	J Value
3040	30	40
Δ3040	3	4
30Δ40	30	4

The first example illustrates the importance of inputting data exactly as specified in the format.

Example 2

```
          READ(1,20)I,J
     20   FORMAT(1X,I2,1X,I2)
```

Input String	I Value	J Value
3040	4	0
Δ3040	30	0
Δ30Δ40	30	40

This example differs from the first in that the "skip one character" indicator appears before each integer field. The first 1X would typically be used to skip past the carriage control character from the original output record. However, if the input string is from a keyboard or cards, the typist must be told exactly how to format his input. The third string in this example is correctly formed.

Example 3

```
          I=123
          J=678
          WRITE (1,100) I,J
     100  FORMAT (1X,I3,1X,I3)
```

```
     Output to Printer             Output to Disk

         123Δ678                     Δ123Δ678
```

Note that the carriage control character to the disk is written exactly as received; to the printer it is converted to a space one line command and does not appear in the printed record.

Example 4

```
          I=123
          J=678
          WRITE (1,100) I,J
     100  FORMAT (I3,1X,I3)
```

```
     Output to Printer             Output to Disk

        (top of form)
        23Δ678                       123Δ678
```

This example is the same as the previous one except that the carriage control character was omitted in the format specification. The output record for the printer is first formed as 123Δ678 (just as it appears on disk). Then, however, the "1" character is treated as a carriage control function. It is replaced by a form feed character so that the printer skips to top of form and prints the record as shown above.

Example 5

```
          I=100
          J=200
          K=300
          WRITE (1,200)I,J,K
     200  FORMAT (1X,I3/'0',I3/'+ΔΔΔ',I3)
```

```
Output to Printer                     Output to Disk

  (single space)
  100                                   Δ100
  (double space)                        0200
  200300                                +ΔΔΔ300
```

Here, three records are written; note the forms control character which appears at the head of each disk record. To the printer, the first record is printed on a new line, the second record is printed after a double space and the third line is printed over the second line.

7.4 FORMAT SPECIFICATION SEPARATORS

Field descriptors in a format specification are generally separated from one another by commas. Slashes (/) may also be used to separate field descriptors. A slash has the additional effect of being a record terminator, causing the input or output of the current record to be terminated and a new record to be initiated. For example:

```
          WRITE (5,40) K,L,M,N,O,P
     40   FORMAT (3O6/I6,2F8.4)
```

is equivalent to:

```
          WRITE (5,40) K,L,M
     40   FORMAT (3O6)
          WRITE (5,50) N,O,P
     50   FORMAT (I6,2F8.4)
```

It is possible to bypass input records or to output blank records (line feed, carriage return sequences) by the use of multiple slashes. If n consecutive slashes appear between two field descriptors, they cause (n-1) records to be skipped on input or (n-1) blank records to be output. (The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.) If n slashes appear at the beginning or end of a format specification, however, they result in n skipped or blank records, because the initial and terminal parentheses of the format specification are themselves a record initiator and record terminator, respectively. An example of the use of multiple record terminators is as follows:

```
        WRITE (5,99)
     99 FORMAT ('1'T51'HEADING LINE'//T51'SUBHEADING LINE'//)
```

The above statements output the following:

```
Column 50, top of page

            HEADING LINE
(blank line)
            SUBHEADING LINE
(blank line)
(blank line)
```

7.5 EXTERNAL FIELD SEPARATORS

A field descriptor such as Fw.d specifies that an Input statement is to read w characters from the external record. If the data field in question contains fewer than w characters, the Input statement would read some characters from the following field unless the short field were padded with leading zeros or spaces. To avoid the necessity of doing so, the data can be read using an implied format specification (see Section 7.7) which, among other things, permits short records to be terminated by a comma, which overrides the field descriptor's field width specification. This practice, called short field termination, is particularly useful when entering data from a terminal keyboard.

7.6 OBJECT TIME FORMAT

Format specifications may be stored in real or double integer arrays, both actual arrays and dummy arrays. Such a format specification (termed an object time format) can be constructed or altered during program execution. The form of a format specification in an array is identical to a FORMAT statement, except that the word FORMAT and the statement label are not present. The initial and terminal parentheses must appear, however. An example of object time format is as follows:

```
          DIMENSION FORRAY(6)
          DATA FORRAY(1),RPAR,FRIG,FMED,FSML/'(',')','F8.2','F9.4','F9.6'/
          .
          .
          .
          DO 20 J=1,5
              DO 18 I=1,5
                 IF (TABLE(I,J) .GE. 100. .OR. TABLE(I,J) .LE. 0.1)
          1        GO TO 14
                 FORRAY(I+1) = FMED
                 GO TO 18
14        IF (TABLE(I,J) .GE. 100.) GO TO 16
                 FORRAY(I+1) = FSML
                 GO TO 18
16        FORRAY(I+1) = FBIG
18        CONTINUE
                 WRITE(5,FORRAY) (TABLE(K,J),K=1,5)
20        CONTINUE
```

In this example, the DATA statement assigns a left parenthesis to the first element of FORRAY and assigns a right parenthesis and three field descriptors to variables for later use. The proper field descriptors are then selected for inclusion in the format specification, based on the magnitude of the individual elements of array TABLE. A right parenthesis is then added to the format specification just before its use by the WRITE statement. Thus, the format specification changes with each iteration of the DO loop.

Here is a second example which establishes an object time format specification (I7, F1Ø.3) entered via a READ statement.

```
      DIMENSION ARRAY(1Ø)
      READ(3,2Ø) (ARRAY(I),I=1,1Ø)
  2Ø  FORMAT (1ØA5)
        .
        .
        .
      READ (3,ARRAY) J,X
```

The first READ statement enters the format into ARRAY. The second READ statement references ARRAY rather than a FORMAT statement number.

7.7 IMPLIED FORMATS

Formatted input/output is normally performed by explicitly referencing in the READ or WRITE statement a FORMAT statement number or an array name, the array containing an object time format specification. However, when a format is not explicitly referenced, as in

```
READ (1,) I,J,K
```

```
        or
    WRITE (3'5,) X(5)
```

then the data format is implied by the following rules.

On input, data items are delimited not by field width but by the occurrence of either a space, comma, or line terminator.[1] For each variable in the input/output list, one item is read, converted to the data type of the variable, and stored. In all cases, a value will be assigned to each variable even if the value cannot be properly converted. For instance, if the value read for an integer variable exceeds the maximum value for integers, the maximum value will be assigned.

If an illegal character appears in any data item, an error message is printed and then input continues. For keyboard input, the user may reenter the erroneous line. The following indicate the acceptable forms of data items under implied format control:

1. Alphanumeric Literals - The rules for forming alphanumeric literals are given in Section 2.2.7.1. If more than five characters are specified within the delimiters of a literal, only the first five are stored; the remainder are discarded. Following the string delimiter (normally a single quote) must be a data item delimiter (space, comma, or line terminator). If more than two characters are expected in an alphanumeric literal, it should be assigned to a double integer variable. The nH text form of Hollerith constant is not acceptable as a substitute for an alphanumeric literal. Some examples of alphanumeric literals are:

```
    'TITLE'
    '57%'
    'A+B='
```

2. Decimal and Octal Constants - The rules of formation for the numeric constants appear in Sections 2.2.1, 2.2.2, 2.2.3, and 2.2.4. An octal integer whose magnitude exceeds 131071_{10} will be considered to be a double integer even if D is not supplied. Some examples follow:

```
    123
    -12.3
    -1.23E+2
    +1Ø.234D+15

    #123
    #1.23
    #D1234567
```

[1] Carriage Return and Altmode are ASCII line terminator characters.

3. Logical Constants - The rules for forming a logical constant appear in Section 7.2.8. Examples of these are:

```
T
F
ΔΔTRUE
.FALSE.
```

On output, the variable name (with subscript if it is an array element) followed by an equals sign followed by the variable's value are output according to the variable's data type.

Variable Type	Implied Output Format
LOGICAL	L1
INTEGER	I7
DOUBLE INTEGER	I12
REAL	G16.8
DOUBLE PRECISION	D20.11

Example

```
LOGICAL   L
DOUBLE INTEGER   DI
DOUBLE PRECISION   DP(S)
I=#100
DI=12345678
R=99.9
DP(1)=9.99999999D3944
L=.TRUE.
WRITE (10,) L,I,DI,R,DP(1)
```

The resulting output would appear as:

```
'L'=T
'I=ΔΔΔΔΔ64
'DI'=ΔΔΔΔ12345678
'R'=ΔΔΔ99.900000ΔΔΔΔ
'DP(1)'=Δ0.9999999900D3945
```

7.8 FORMAT CONTROL INTERACTION WITH INPUT/OUTPUT LISTS

Format control is initiated with the beginning of execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the FORMAT statement or format array or the implied format specification. Both the I/O list and the format specification, except for the effects of repeat counts, are interpreted from left to right.

If the I/O statement contains an I/O list, at least one field descriptor of a type other than H, X, T or P must exist in the format specification.

When a formatted input statement is executed, it reads one record from the specified device and initiates format control; thereafter, additional records may be read as indicated by the format specification. Format control demands that a new record be input whenever a slash is encountered in the format specification, or when the last outer right parenthesis of the format specification is reached and I/O list elements remain to be filled. Any remaining characters in the current record are discarded at the time the new record is read.

When a formatted output statement is executed, it transmits a record to the specified device as format control terminates. Records may also be output during format control if a slash appears in the format specification or if the last outer right parenthesis is reached and more I/O list elements remain to be transmitted.

Each field descriptor of types I, O, F, E, D, G, L, A, and R corresponds to one element in the I/O list. No list element corresponds to an H, X, T, or alphanumeric literal field descriptor. In the case of H and alphanumeric literal field descriptors, data transfer takes place directly between the external record and the format specification.

When format control encounters an I, O, F, E, D, G, L, A, or R field descriptor, it determines if a corresponding element exists in the I/O list. If so, format control transmits data, appropriately converted to or from external format, between the record and the list element, then proceeds to the next field descriptor (unless the current one is to be repeated). If there is no associated list element, format control terminates.

When the last outer right parenthesis of the format specification is reached, format control determines whether or not there are more I/O list elements to be processed. If not, format control terminates. If additional list elements remain, however, the current record is terminated, a new one initiated, and format control reverts to the rightmost top-level group repeat specification (the one whose left parenthesis matches the next-to-last right parenthesis of the format specification). If no group repeat specification exists in the FORMAT statement or format array, format control returns to the initial left parenthesis of the format specification. Data transfer continues from that point.

7.9 SUMMARY OF RULES FOR FORMAT STATEMENTS

The following is a summary of the rules pertaining to the construction and use of the FORMAT statement or format array and its components, and to the construction of the external fields and records with which a format specification communicates. The rules for implied format I/O are covered in Section 7.7.

7.9.1 General

1. A FORMAT statement must always be labeled.

2. In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than zero. The repeat count may be omitted; the field width specification must be present.

3. In a field descriptor such as Fw.d, the term d must be an unsigned integer constant. It must be present in F, E, D, and G field descriptors even if it is zero. The decimal point must also be present. The field width specification, w, must be greater than or equal to d.

4. In a field descriptor such as nHc1c2 ... cn, exactly n characters must be present. Any printable ASCII character, including space, may appear in this field descriptor (an alphanumeric literal field descriptor follows the same rule).

5. In a scale factor of the form nP, n must be a signed or unsigned integer constant. Use of the scale factor applies to F, E, D, and G field descriptors only. Once a scale factor has been specified, it applies to all subsequent real or double precision field descriptors in that format specification until another scale factor appears; an explicit 0P specification is required to reinstate a scale factor of zero.

6. No repeat count is permitted in H, X, T or character constant descriptors unless those field descriptors are enclosed in parentheses and treated as a group repeat specification.

7. If an I/O list is present in the associated I/O statement, the format specification must contain at least one field descriptor of a type other than H, X, T or alphanumeric literal.

8. A format specification in an array must be constructed identically to a format specification in a FORMAT statement, including the initial and terminal parentheses.

When a format array name is used in place of a FORMAT statement label in an I/O statement, that name must not be subscripted.

7.9.2 Input

1. An external input field with a negative value must be preceded by a minus symbol; a positive-value field may optionally be preceded by a plus sign.

2. An external field whose input conversion is governed by an I field descriptor must have the form of a decimal integer constant. An external field input under control of an O field descriptor must have the form of an octal integer constant without a leading pound sign (#) or D designator. Neither may contain a decimal point or an exponent.

3. An external field whose input conversion is handled by an F, E, or G field descriptor must have the form of an integer constant or a real or double precision constant. It may contain a decimal point and/or an E or D exponent field.

4. If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the associated real or double precision field descriptor.

5. If an external field contains an exponent, it causes the scale factor (if any) of the associated field descriptor to be inoperative for the conversion of that field.

6. The field width specification must be large enough to accommodate, in addition to the numeric character string of the external field, any other characters that may be present (algebraic sign, decimal point, and/or exponent).

7. When data are read via implied format, a comma is the only character that is acceptable for use as an external field separator. It is used to terminate input of fields that are shorter than the number of characters expected, or to designate null (zero-length) fields.

7.9.3 Output

1. A format specification must not demand the output of more characters than can be contained in the external record (for example, a line printer record cannot contain more than 133 characters including the carriage control character).

2. The field width specification, w, must be large enough to accommodate all characters that may be generated by the output conversion, including an algebraic sign,

decimal point, and exponent (the field width specification in an E field descriptor, for example, should be large enough to contain (d+7) characters).

3. The first character of a record output to a printing device is used for carriage control; it is never printed. The first character of such a record should be a space, 0, 1, or +. Any other character is treated as a space and is deleted from the record.

CHAPTER 8

SUBPROGRAMS

A subprogram is a program which is invoked by name from other programs (with the exception of an arithmetic statement function) whenever the operations it performs are required. It is a convenient and efficient means for encoding frequently used or complex operations since the statements appear only once in the object program regardless of the number of times they are used. In addition, a subprogram may be designed to handle a variety of different values which may be transmitted as arguments whenever the subprogram is invoked. The process of establishing a subprogram is referred to as subprogram definition; the statements to be executed are referred to as the body of the subprogram; the process of invoking the subprogram and transmitting arguments is referred to as a subprogram call.

FORTRAN subprograms are divided into two general classes: those that are written by the user and those that are supplied by the FORTRAN system. User-written subprograms are grouped into the categories of functions, which includes both arithmetic statement functions and FUNCTION subprograms, subroutines, and BLOCK DATA subprograms.

8.1 USER-WRITTEN SUBPROGRAMS

One difference between functions and subroutines is that control is transferred to a function by means of a function reference while control is passed to a subroutine by a CALL statement. A function reference is simply the name of the function, together with its arguments, appearing in an expression.

A second difference is that a function returns a value to the calling program whenever a normal RETURN statement is executed (without a statement return variable). Both functions and subroutines may return additional values via assignment to their arguments.

Arguments are represented in two ways: as dummy arguments and as actual arguments. Dummy arguments appear in the FUNCTION statement, SUBROUTINE statement, or arithmetic statement function definition and are used to represent the value of the corresponding actual argument. Actual arguments appear in the function reference or CALL statement and provide actual values to be used for computation. The actual and dummy arguments become associated at the time control is transferred to the subprogram. Actual arguments may be constants, variables, array names, array elements, subprogram names, or expressions.

8.1.1 Arithmetic Statement Function (ASF)

An arithmetic statement function is a computing procedure defined by a single statement, similar in form to an arithmetic assignment statement. The appearance of a reference to the function within the same program unit causes the computation to be performed and the resulting value made available to the expression in which the ASF reference appears.

The statement that defines an arithmetic statement function is non-executable and appears in the following general form:

```
f (p[,p]...)=e
```

f   is a symbolic name of the ASF in the same form as a variable name

p   is a symbolic name of a dummy argument in the same form as a variable name

e   is an arithmetic expression that defines the computation to be performed by the ASF

A function reference to an ASF is executable and takes the form:

```
f (p[,p]...)
```

where f is the name of the ASF, and each p is an actual argument. An actual argument may be any arithmetic expression.

When a reference to an arithmetic statement function appears in an expression, the values of the actual arguments are associated with the dummy arguments in the ASF definition. The expression in the defining statement is then evaluated and the resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of an ASF is determined either implicitly by the initial letter of the name or explicitly by appearance in a data type declaration statement.

The expression, e, which defines the value of an ASF may include dummy variables, ordinary variables, non-Hollerith constants, and previously-defined external functions and arithmetic statement functions. The dummy variables used to define the ASF may also appear in a specification statement, other than COMMON, DATA and EQUIVALENCE, but in no other context. Up to 20 dummy arguments may appear in a single definition. The definition of an ASF must not contain a reference to itself.

Any dimensioning information associated with the dummy argument name will be ignored in the ASF. The name of the ASF, however, cannot be used to represent any other entity within the same program unit.

Any reference to an ASF must appear in the same program unit as the definition of that function.

An ASF reference must appear as, or be part of, an expression; it must not be used as a variable or array name on the left of an equal sign.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments. Values must have been assigned to them before control is transferred to the arithmetic statement function.

Examples

ASF Definitions

```
VOLUME(RADIUS) = 4.189*RADIUS**3
SINH(X) = (EXP(X)-EXP(-X))*0.5
AVG(A,B,C,3.) = (A+B+C)/3.        (Invalid; constant as dummy
                                   argument not permitted)
```

In the second example, the function EXP is an exponential function supplied in the FORTRAN Library. It raises the value of the mathematical constant e (approximately 2.71828) to the power of the argument.

ASF References

```
AVG(A,B,C) = (A+B+C)/3.          (Definition)
    .
    .
GRADE = AVG(TEST1,TEST2,XLAB)
IF (AVG(P,D,Q).LT.AVG(X,Y,Z)) GO TO 300
FINAL = AVG(TEST3,TEST4,LAB2)     (Invalid; data type of third
                                  argument does not agree with
                                  dummy argument)
```

8.1.2 FUNCTION Subprogram

A FUNCTION subprogram is a program unit that consists of a FUNCTION statement followed by a series of statements that define a computing procedure. Control is transferred to a FUNCTION subprogram by a function reference and returned to the calling program unit by a RETURN statement.

A FUNCTION subprogram returns a single value to the calling program unit by assigning that value to the function's name. The data type of the value returned is determined by the function's name.

The FUNCTION statement appears in the following general form:

```
[typ] FUNCTION nam p[,p]...)
```

typ is a type specifier, such as INTEGER, REAL, etc.

nam is a symbolic name of the function in the same form as a variable name

p is a symbolic name of a dummy argument, which must be an unsubscripted symbolic name

A function reference that transfers control to a FUNCTION subprogram takes the general form:

```
nam (p[,p]...)
```

where nam is the symbolic name of the function to receive control, and each p is an actual argument, which may be any valid expression or the name of an external function or subroutine. There may be up to 20 arguments.

When control is transferred to a FUNCTION subprogram, the values supplied by the actual arguments are associated with the dummy arguments in the FUNCTION statement. The statements in the subprogram are then executed, using those values. The name of the function must be assigned a value before a RETURN statement is executed in that function. When control is returned to the calling program unit, the value thus assigned to the function's name is made available to the expression that contains the function reference, and is used to complete the evaluation of that expression. A function reference may appear only in an expression.

The type of a function name may be specified implicitly, explicitly in the FUNCTION statement, or explicitly in a type declaration statement.

The FUNCTION statement must be the first statement of a function subprogram. It must not be labeled.

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

A FUNCTION subprogram must not contain a SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement other than the initial statement of the subprogram.

If an actual argument is a constant, subprogram name, or statement label, the function must not attempt to alter the value of the corresponding dummy argument.

A FUNCTION subprogram may contain references to other subprograms, but not to itself; recursion is not allowed.

Actual arguments must agree in number, order, and data type with the dummy arguments of the function. The type of the function name as defined in the FUNCTION subprogram must be the same as the type of the function name in the calling program unit.

They need not all be of the same type and need not be of the same type as the function name.

When arrays are involved, DIMENSION statements (or the equivalent) must appear for the actual arrays in the calling program and for the corresponding dummy arrays in the body of the subprogram. If the dummy variable is not used as an array name (discussed below) then a DIMENSION statement is not used in the subprogram, although it is still necessary in the calling program.

For example:

Calling Program

```
DIMENSION SQUARE (10,10)
BASE = 1.0E-05+AVG(SQUARE,100)
```

Subprogram

```
FUNCTION AVG(A,I)
DIMENSION A(I)
```

In the above example, the FUNCTION subprogram to compute an average value of all the elements of an array uses two dummy arguments; one is the dummy array name and the other is the dummy array size. Both are used in a dimension statement within the function subprogram. Using an adjustable array in this fashion allows the subprogram to know each time it is called how large the array is. Note that the actual array has two dimensions but that the dummy array has one. The number of subscripts in the dummy array need not correspond with the number of subscripts in the actual array. Also, the dummy array size need not be the same as that of the actual array. If it is larger, however, one runs the risk of accessing undefined data unless it is known what follows the actual array in core storage due to COMMON and/or EQUIVALENCE statements.

If the actual argument in a call or reference to a subprogram is an array name, the corresponding dummy argument need not be an array name. If it is, the subprogram will have access to the entire array (provided that the dimensions are equivalent). If it is not, it is a simple variable which becomes equated with the first element of the array.

For example:

Calling Program

```
DIMENSION X(200)
ANS=F(X)
```

Subprogram

```
FUNCTION F(A)
F=A**2+0.9
```

If the value stored in the first element of array X were X(1)=0.1, then the evaluation of the function would be equivalent to F(X(1))=0.1 **2+0.9 which becomes 0.91.

If the actual argument is not an array name, but is a simple variable, array element, or constant, the dummy variable associated with it may be an array name or simply a variable name. If the dummy argument is an array name, then the actual argument, if it is a variable name or array element, is treated as the first element of the dummy array. There are two practical cases where this makes sense. The first is where the actual variable is equated with an element (not necessarily the first) of an actual array or the actual element is passed in the call. For instance,

Calling Program

```
DIMENSION  A(100)
ANSWER=SUM(A(1))
      :
ANSWER=SUM(A(51))
```

Subprogram

```
FUNCTION  SUM(WINDOW)
DIMENSION WINDOW(50)
SUM=0
DO 1 I=1,50
SUM=SUM+WINDOW(I)
RETURN
END
```

Here a subprogram is called twice to compute the partial sum of elements within a 50-word window in an array. The actual argument transmitted is not an array name but an array element; however, the dummy argument is an array name. The second practical case where the dummy variable is an array name but the actual argument is a variable or array element occurs when the passed argument is a variable equated to an element of an array or is an element in a common block in which more elements follow. The following example shows a permissible sequence:

Calling Program

```
COMMON  A(10), B(20)
EQUIVALENCE  (C,A(6))
R=F(C)
```

Subprogram

```
      FUNCTION  F(X)
      DIMENSION  X(25)
```

This sequence allows function F to access the last five elements of array A and all twenty elements of array B as a single array X.

Finally, the case where the actual argument is a constant and the dummy argument is an array name, is meaningless and errors are certain to occur.

The logical termination of a FUNCTION subprogram is signalled by a RETURN statement. The physical end is indicated by an END statement.

Example

```
      FUNCTION ROOT(A)
      X = 1.0
    2 EX = EXP(X)
      EMINX = 1./EX
      ROOT = ((EX+EMINX)*.5+COS(X)-A)/((EX - EMINX)*.5-SIN(X))
      IF (ABS(X-ROOT).LT.1.E-6) RETURN
      X = ROOT
      GO TO 2
      END
```

The function in this example uses the Newton-Raphson iteration method to obtain the root of the function:

$$F(X) = \cosh(X) + \cos(X) - A = 0$$

where the value of A is passed as an argument. The iteration formula for this root is:

$$Xi+1 = Xi - \frac{\text{Cosh } (Xi)+\cos(Xi)-A}{\sinh\ (Xi)-\sin(Xi)}$$

which is repeatedly calculated until the difference between Xi and Xi+1 is less than 1 x 10^{-6}. The function makes use of the FORTRAN Library functions EXP, SIN, COS, and ABS.

8.1.3 SUBROUTINE Subprogram

A SUBROUTINE subprogram is a program unit that consists of a SUBROUTINE statement followed by normal FORTRAN statements with the added requirement that the body of the SUBROUTINE contain at least one RETURN statement. The RETURN statement is the logical end of a subroutine; whereas, the END statement is the physical END of the subroutine.

Control is transferred to a subroutine by a CALL statement and returned to the calling program unit by a RETURN statement.

The SUBROUTINE statement appears in the following form:

```
SUBROUTINE nam [(p[,p]...)]
```

nam is the symbolic name of the subroutine, of the same form as a variable name.

p is a dummy argument, which must be an unsubscripted symbolic name

The form of the CALL statement is described in Section 4.5, and the RETURN statement in Section 4.6. The ability to have several return points is explained in these sections.

When control is transferred to the subroutine, the values supplied by the actual arguments (if any) in the CALL statement are associated with the corresponding dummy arguments (if any) in the SUBROUTINE statement, making those values available to the subprogram.

The SUBROUTINE statement must be the first statement of a subroutine; it must not have a statement label.

The argument list in a SUBROUTINE statement may contain up to 2Ø dummy arguments, or none. Dummy arguments must not appear in a COMMON, DATA, or EQUIVALENCE statement within the subprogram.

A subroutine cannot contain a FUNCTION statement, a BLOCK DATA statement, or a SUBROUTINE statement other than the initial statement of the subprogram. A CALL statement in a subroutine may transfer control to other subroutines but must not transfer control to the subroutine of which it is a part.

The name of the subroutine must not be used as a variable in the calling program unit and may not appear within any statement in the body of the subroutine. Because the subroutine name, unlike a FUNCTION subprogram name, is not associated with a data value to be returned to the caller, the subroutine name is not typed (see Section 2.0).

Statements in a subroutine may establish or redefine values for any dummy argument whose associated actual argument is not a constant, expression, subprogram name, or statement label. In such a case, the dummy argument in the body of the subroutine must appear either on the left-hand side of an arithmetic assignment statement or in a variable list as part of a READ statement.

Actual arguments in a CALL statement must agree in number, order, and data type with the dummy arguments in the corresponding SUBROUTINE statement.

When arrays are involved, DIMENSION statements (or the equivalent) must appear for the actual arrays in the calling program and for the corresponding dummy arrays in the body of the subprogram. If the dummy variable is not used as an array name (discussed below) then a DIMENSION statement is not used in the subprogram, although it is still necessary in the calling program.

The number of subscripts in the dummy array and its total size need not match that of the actual array. Analogous examples in Section 8.1.2 on FUNCTION subprograms explain the consequences.

If the actual argument in a call or reference to a subprogram is an array name, the corresponding dummy argument need not be an array name. If it is, the subprogram will have access to the entire array (provided that the dimensions are equivalent). If it is not, it is a simple variable which becomes equated with the first element of the array.

If the actual argument is not an array name, but is a simple variable, array element or constant, the dummy variable associated with it may be an array name or simply a variable name. If the dummy argument is an array name, the actual argument, if it is a variable name or array element, is treated as the first element of the dummy array.

Examples

Main Program

```
      DIMENSION A(100), B(100), C(100)
                 .
                 .
                 .
      CALL MERGE (A,B,C, 100)
```

SUBROUTINE Subprogram

```
           SUBROUTINE MERGE (X,Y,Z,I)
           DIMENSION X(I), Y(I), Z(I)
           DO 10 J=1, I
           Z(J)=Y(J)
10         IF  (X(J).GE. Y(J))    Z(J) =X (J)
           RETURN
           END
```

In this example, a subroutine is called to merge two arrays into a third, keeping the largest value of each element.

Main Program

```
      C    .
      C    .
           COMMON NFACES,EDGE,VOLUME
      C    .
      C    .
      C    .
      C    .
      C    .
      C    .
           READ (5,65) NFACES,EDGE
      65   FORMAT(I2,F8.5)
           CALL PLYVOL
      C    .
      C    .
      C    .
      C    .
      C    .
      C    .
           STOP
           END
```

Subprogram

SUBROUTINE Subprogram

```
          SUBROUTINE PLYVOL
          COMMON NFACES,EDGE,VOLUME
          CUBED = EDGE**3
          GO TO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,5,6,5,6), NFACES
1         VOLUME = CUBED * 0.11785
          RETURN
2         VOLUME - CUBED
          RETURN
3         VOLUME = CUBED * 0.47140
          RETURN
4         VOLUME = CUBED * 7.66312
          RETURN
5         VOLUME = CUBED * 2.18170
          RETURN
6         WRITE (7,100) NFACES
100       FORMAT(' NO REGUALR POLYHEDRON HAS ',I3,' FACES.'/)
          RETURN
          END
```

The subroutine in this example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses the computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and to transfer control to the proper procedure for calculating the volume. If the number of faces of the body is other than 4, 6, 8, 12, or 20, the subroutine displays an error message on the user's terminal. Note that in this example the subroutine arguments, rather than being passed in the CALL statement, are stored in common which can be accessed by both the main program and subroutine.

8.1.4 ENTRY Statement

The ENTRY statement provides multiple entry points within an external subprogram. It is not executable and can appear within a function or subroutine subprogram after all specification statements and statement function definitions. Execution begins with the first executable statement following the ENTRY statement.

An ENTRY statement may appear in the following form:

```
ENTRY nam (p[,p]...)
```

nam is the entry name
p is a dummy argument.

The entry name must not be referenced from inside the program unit in which it appears.

The ENTRY statement cannot appear within a DO loop.

Entry names appearing in ENTRY statements within SUBROUTINE subprograms must be referenced by CALL statements, and entry names appearing in ENTRY statements within FUNCTION subprograms must be referenced as external function references.

A function entry name can appear in a type statement.

Any entry name can appear in an EXTERNAL statement and be used as an actual argument; the entry name in an ENTRY statement cannot be a dummy argument.

Entry names cannot appear in executable statements that physically precede the appearance of the entry name in an ENTRY statement.

The order, number, type and names of the dummy arguments in an ENTRY statement can be different from the order, number, type and names of the dummy arguments in the FUNCTION statement, SUBROUTINE statement, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement. The dummy arguments of an ENTRY may appear in the body of the subprogram prior to the ENTRY statement only if they are also arguments of a prior ENTRY or of the SUBROUTINE or FUNCTION definition.

8.1.4.1 ENTRY in Function Subprogram - The function and entry names are not required to be the same type, but at the execution of a RETURN statement, the name used to reference the function subprogram must be defined. Note that an entry name cannot appear in executable statements that precede the appearance of the entry name in an ENTRY statement.

8.1.4.2 ENTRY and Array Declarator Interaction - A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any of the variables appearing in the adjustable array declarator are not currently associated with an actual argument or are not in a COMMON block. Although an adjustable array name and its adjustable dimensions may appear as dummy arguments in the ENTRY statement, no size adjustment of the array is performed. Such adjustable arrays must first be dimensioned following the SUBROUTINE or FUNCTION definition, and the array cannot be referenced unless entry is first made at the SUBROUTINE or FUNCTION definition. Note that there is no retention of argument association between one reference of a subprogram and the next reference of that subprogram. Consider the following example:

```
SUBROUTINE S(A,I,J)
DIMENSION A(I)
A(I) = J
ENTRY S1(I,A,2)
A(I) = A(I) + 1
ENTRY S2
RETURN
END
```

If B is a real array with 10 elements, as in

```
DIMENSION B(10)
```

then the statement

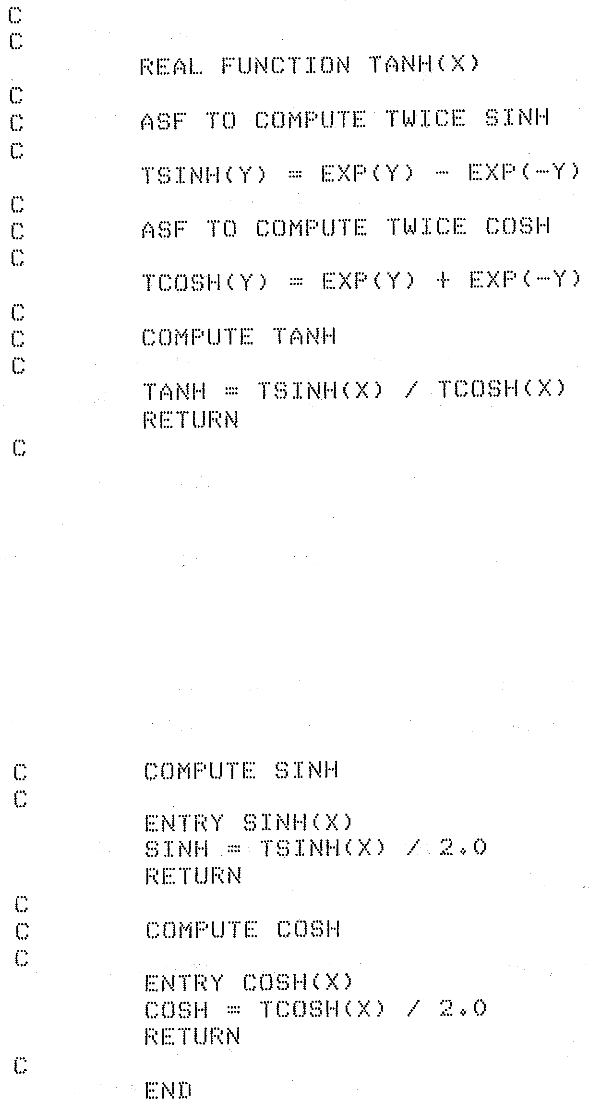
```
CALL S(B,2,3)
```

would set B(2) = 4 and the statement

```
CALL S1(5,B)
```

would increment B(5) by 1.

A single function routine that provides the hyperbolic functions sinh, cosh, and tanh appears in Figure 8-1.



```
C
C
        REAL FUNCTION TANH(X)
C
C       ASF TO COMPUTE TWICE SINH
C
        TSINH(Y) = EXP(Y) - EXP(-Y)
C
C       ASF TO COMPUTE TWICE COSH
C
        TCOSH(Y) = EXP(Y) + EXP(-Y)
C
C       COMPUTE TANH
C
        TANH = TSINH(X) / TCOSH(X)
        RETURN
C




C       COMPUTE SINH
C
        ENTRY SINH(X)
        SINH = TSINH(X) / 2.0
        RETURN
C
C       COMPUTE COSH
C
        ENTRY COSH(X)
        COSH = TCOSH(X) / 2.0
        RETURN
C
        END
```

Figure 8-1
A Single Function Subprogram to Provide the Hyperbolic Functions SINH, COSH, and TANH

8.1.5 BLOCK DATA Subprogram

A BLOCK DATA subprogram is used to assign initial values to entities in a single labeled common block, at the same time establishing and defining that block. It consists of a BLOCK DATA statement followed by a series of nonexecutable statements (specification statements).

The BLOCK DATA statement appears in the form:

```
          BLOCK DATA
```

The statements allowed in a BLOCK DATA subprogram are: Type Declaration, IMPLICIT, DIMENSION, COMMON, EQUIVALENCE, and DATA statements.

The BLOCK DATA subprogram functions at compilation time only. The specification statements in the subprogram establish and define a common block, assign variables and arrays to that block, and place initial data in those components.

The BLOCK DATA subprogram is the only way in which components in labeled common blocks can be initialized. Components in blank common can never be initialized.

A BLOCK DATA statement must be the first statement of a BLOCK DATA subprogram. It must not be labeled.

A BLOCK DATA subprogram must not contain any executable statements.

If any entity in a labeled common block is initialized in a BLOCK DATA subprogram, a complete set of specification statements to establish the entire block must be present, even though some of the components in the block do not appear in a DATA statement.

More than one common block may be defined in a BLOCK DATA subprogram; however, DATA statements must be used only to initialize the last block. To initialize more than one named common block, one must use more than one BLOCK DATA subprogram. The following example, although it will not cause a compilation error, will not function properly.

```
          BLOCK DATA
          COMMON /N1/I(10)/N2/J(20)
          DATA I/10*1/,J/20*2/
          END
```

Example

```
BLOCK DATA
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U,W,X,Y
DATA R/1.0,2*2.0/,T/.FALSE./,U/0.214537D-7/,W/.TRUE./,Y/3.5/
END
```

8.2 FORTRAN LIBRARY FUNCTIONS

The FORTRAN library functions are listed in Table B-1. In order to use a library function in any FORTRAN program, it is only necessary to use the symbolic name of the function, together with the required data references (arguments) upon which the function is to act. The value obtained from the execution of the function is made available to the containing expression. For example,

```
J = IFIX(2*SQRT(C))
```

where SQRT is an external library function and IFIX is an intrinsic library function.

The data type of each library function is predefined as described in Table B-1. Arguments passed to these functions may consist of subscripted or simple variable names, expressions, constants, arithmetic functions. Arguments to these functions must correspond to the type indicated in Table B-1.

APPENDIX A

CHARACTER CODES

A.1 FORTRAN CHARACTER SET

The FORTRAN character set consists of:

1. The letters A through Z
2. The numerals 0 through 9
3. The following special characters:

Character	Name
Δ	Space or blank or tab
⇥	Tab
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe (Single Quote)
"	Double Quote
$	Dollar Sign
[	Left (Open) Bracket
]	Right (Close) Bracket
:	Colon
;	Semicolon
#	Sharp Sign (Pound Sign)
@	At sign

Other printable characters may appear in a FORTRAN statement only as part of a Hollerith constant, alphanumeric literal, or a comment.

A.2 ASCII CHARACTER CODE

The following table shows the correspondence between the PDP-15 64-character graphic subset of ASCII and the DEC 029/026 Hollerith codes. Both 029 and 026 codes are identical for numeric and alphabetic characters but vary for symbol representation. The 029 code, except as indicated by brackets [], is a subset of the standard Hollerith punched card code specified in ANSI standard X3.26-1970.

	PARITY				PARITY		
CHARACTER	ASCII	DEC029	DEC026	CHARACTER	ASCII	DEC029	DEC026
SPACE	240	NONE	NONE	@	300	8 4	8 4
!	041	11 8 2	12 8 7	A	101	12 1	12 1
"	042	8 7	0 8 5	B	102	12 2	12 2
#	243	8 3	0 8 6	C	303	12 3	12 3
$	044	11 8 3	11 8 3	D	104	12 4	12 4
%	245	0 8 4	0 8 7	E	305	12 5	12 5
&	246	12	11 8 7	F	306	12 6	12 6
'	047	8 5	8 6	G	107	12 7	12 7
(	050	12 8 5	0 8 4	H	110	12 8	12 8
)	251	11 8 5	12 8 4	I	311	12 9	12 9
*	252	11 8 4	11 8 4	J	312	11 1	11 1
+	053	12 8 6	12	K	113	11 2	11 2
,	254	0 8 3	0 8 3	L	314	11 3	11 3
-	055	11	11	M	115	11 4	11 4
.	056	12 8 3	12 8 3	N	116	11 5	11 5
/	257	0 1	0 1	O	317	11 6	11 6
0	060	0	0	P	120	11 7	11 7
1	261	1	1	Q	321	11 8	11 8
2	262	2	2	R	322	11 9	11 9
3	063	3	3	S	123	0 2	0 2
4	264	4	4	T	324	0 3	0 3
5	065	5	5	U	125	0 4	0 4
6	066	6	6	V	126	0 5	0 5
7	267	7	7	W	327	0 6	0 6
8	270	8	8	X	330	0 7	0 7
9	071	9	9	Y	131	0 8	0 8
:	072	8 2	11 8 2	Z	132	0 9	0 9
;	273	11 8 6	0 8 2	[	333	12 8 2	11 8 5
<	074	12 8 4	12 8 6	\	134	11 8 7	8 7
=	275	8 6	8 3	]	335	0 8 2	12 8 5
>	276	0 8 6	11 8 6	↑ or ^	336	11 8 7	8 5
?	077	0 8 7	12 8 2	← or _	137	0 8 5	8 2

1. ASCII codes 00-37 and 140-177 have no corresponding codes in the DEC 026 and 029 Hollerith sets and therefore, are not presented here.

2. ALT MODE is simulated by a 12-8-1 punch (multiple punch A8).

3. End-of-file corresponds to a 12-11-0-1 punch (multiple punch A0-1).

APPENDIX B

FORTRAN LANGUAGE SUMMARY

B.1 EXPRESSION OPERATORS

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

Type	Operator		Operates Upon
Arithmetic	**	exponentiation	arithmetic or logical constants, variables, array elements, function references and expressions
	-	unary minus	
	*,/	multiplication, division	
	+,-	addition and subtraction (but not unary minus)	
Relational	.GT.	greater than	arithmetic or logical constants, variables, array elements, function references, and 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	logical or integer constants, variables, array elements, function references and expressions
	.AND.	A.AND.B is true if and only if A and B are both true	
	.OR.	A.OR.B is true if and only if either A or B or both are true	
	.XOR.	A.XOR.B is true if and only if A is true and B is false, or B is true and A is false.	

B.2 STATEMENTS

The following summary of statements available in the XVM FORTRAN language defines the general format for the statement. If more detailed information is needed, the reader is referred to the Section(s) in this manual dealing with that particular statement.

Statement Formats	Effect	Manual Section
Arithmetic/Logical Assignment		
$v_1=v_2=...=v_n=e$		3.1
v_i	is a variable name or an array element name	
e	is an expression	
	The value of the arithmetic or logical expression is assigned to each variable, from right to left.	
Arithmetic Statement Function		
f(p[,p]...)=e		8.1.1
f	is a symbolic name	
p	is a symbolic name	
e	is an arithmetic expression	
	Creates a user-defined function having the variables p as dummy arguments. When referenced, the expression is evaluated using the actual arguments in the function call.	
ASSIGN s TO v		3.3
s	is an executable statement label	
v	is an integer variable name	
	Associate the statement number s with the integer variable v for later use in an assigned GO TO statement.	
BACKSPACE u		6.8.2
u	is an integer variable or constant	
	The currently open file on logical unit number u is backspaced one record.	

FORTRAN Language Summary

Statement Formats — Effect — Manual Section

BLOCK DATA — 8.1.5

Specifies the subprogram which follows as a BLOCK DATA subprogram.

CALL s[(a[,a]...)] — 4.5

s — is a subprogram name

a — is an expression, a procedure name, or an array name

Calls the SUBROUTINE subprogram with the name specified by s, passing the actual arguments a to replace the dummy arguments in the SUBROUTINE definition.

COMMON [/[cb]/] nlist [/[cb]/ nlist]... — 5.4

cb — is a common block name

nlist — is a list of one or more variable names, array names, or array declarators separated by commas.

Reserves one or more blocks of storage space under the name specified to contain the variables associated with that block name.

CONTINUE — 4.4

Causes no processing, and is most often used to terminate DO loops.

DATA nlist/clist/[,nlist/clist/]... — 5.7

nlist — is a list of one or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

clist — is a list of one or more constants separated by commas, each optionally preceded by j*, where j is a nonzero, unsigned integer constant.

Causes elements in the list of values to be initially stored in the corresponding elements of the list of variable names.

FORTRAN Language Summary

Statement Formats — Effect — Manual Section

DECODE (c,a[f][,ERR=s])[list] — 6.9

c — is an integer expression

a — is an array name

f — is a FORMAT statement label or array name

s — is a statement label

list — is an I/O list

Changes the elements in the I/O list from ASCII into the desired internal format; c specifies the number of characters, f specifies the format, and a is the name of an array containing the ASCII characters to be converted.

DIMENSION a(d)[,a(d)]... — 5.3

a(d) — is an array declarator

Specifies storage space requirements for arrays.

DO s i = vl,v2[,[-]v3] — 4.3

s — is the label of an executable statement

i — is an integer variable name

vn — are integer expressions

1. Set i = vl
2. Execute statements through statement number s
3. Evaluate i = i±v3
4. Repeat 2 through 3 for INT((v2 - vl)/v3) iterations.

ENCODE (c,a[f][,ERR=s])[list] — 6.9

c — is an integer expression

a — is an array name

f — is a FORMAT statement label or an array name

s — is a statement label

list — is an I/O list

Statement Formats — Effect — Manual Section

ENCODE (cont.)

Changes the elements in the list of variables into ASCII format; c specifies the number of characters in the buffer, f specifies the format statement number, and a is the name of the array to be used as a buffer.

END — 4.9

Specifies the physical end of a program unit.

ENDFILE u — 6.8.3

u — is an integer variable or constant

An end-file record is written on logical unit u, following output statements to that unit.

ENTRY nam (p[,p]...) — 8.1.4

nam — is a symbolic name

p — is a symbolic name

Defines an alternate entry point within a SUBROUTINE or FUNCTION subprogram.

EQUIVALENCE (nlist)[,(nlist)]... — 5.5

nlist — is a list of two or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

Each of the names (nlist) within a set of parentheses is assigned beginning at the same storage location.

EXTERNAL v[,v]... — 5.6

v — is a procedure name

Informs the system that the names specified are those of FUNCTION or SUBROUTINE programs.

EXTERNAL v[,v]... — 5.6

v — is a procedure name

Informs the system that the names specified are user-defined.

FORTRAN Language Summary

Statement Formats — Effect — Manual Section

FORMAT (field specification,...) — 7.1 - 7.8

Describes the format in which one or more records are to be transmitted; a statement label must be present.

[typ] FUNCTION nam(p[,p]...) — 8.1.2

typ is a type specifier

nam is a symbolic name

p is a symbolic name

Begins a FUNCTION subprogram, indicating the program name and any dummy argument names, p. An optional type specification can be included.

GO TO s — 4.1.1

s is an executable statement label

(Unconditional GO TO) Transfers control to statement number s.

GO TO (slist),v — 4.1.2

slist is a list of one or more executable statement labels separated by commas.

v is an integer variable

(Computed GO TO) Transfers control to the statement label specified by the value v. (If v=1 control transfers to the first statement label. If v=2 it transfers to the second statement label, etc.) If v is less than 1 or greater than the number of statement labels present, no transfer takes place.

GO TO v[,(slist)] — 4.1.3

v is an integer variable name

slist is a list of one or more executable statement labels separated by commas

(Assigned GO TO) Transfers control to the statement most recently associated with v by an ASSIGN statement.

Statement Formats	Effect	Manual Section

IF (e) v1,v2,v3 — 4.2.1

e — is an expression

vi — are executable statement labels or variables to which statement labels have been ASSIGNed.

(Arithmetic IF) Transfers control to statement number vi depending upon the value of the expression. If the value of the expression is less than zero, transfer to v1; if the value of the expression is equal to zero, transfer to v2; if the value of the expression is greater than zero, transfer to v3.

IF (e) st — 4.2.2

e — is an expression

st — is any executable statement except a DO or a logical IF statement

(Logical IF) Executes the statement if the logical expression is true.

IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]... — 5.1

typ — is a data type specifier

a — is either a single letter, or two letters in alphabetical order separated by a dash (i.e., x-y)

The elements a represent single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that type.

PAUSE [disp] — 4.7

disp — is an octal integer constant.

Suspends program execution and prints the display, if one is specified.

PRINT — See WRITE, for which PRINT is a synonym. — 6.4.5

Statement Formats — Effect — Manual Section

READ (u,[f][,END=s][,ERR=s])[list] — 6.4.1

u — is an integer variable or constant

f — is a FORMAT statement label or an array name

s — is an executable statement label

list — is an I/O list

(Formatted Sequential) Reads at least one logical record from device u according to format specification f and assigns values to the variables in the optional list.

READ (u'r,[f][,ERR=s])[list] — 6.6.1

u — is an integer variable or constant

r — is an integer expression

f — is a FORMAT statement label or an ARRAY name

s — is an executable statement label

list — is an I/O list

(Formatted Direct Access READ) Reads record number r from unit u and assigns values to the elements of the list according to format f.

READ(u[,END=s][,ERR=s])[list] — 6.3.1

u — is an integer variable or constant

s — is an executable statement label

list — is an I/O list

(Unformatted Sequential READ) Reads one unformatted record from device u, assigning values to the variables in the optional list.

READ(u'r[,ERR=s])[list] — 6.5.1

u — is an integer variable or constant

r — is an integer expression

s — is an executable statement label

list — is an I/O list

Statement Formats — Effect — Manual Section

(Unformatted Direct Access READ) Reads record r from logical unit u, assigning values to the variables in the optional list.

RETURN [v] — 4.6

v — is an integer variable

Returns control to the calling program from the current subprogram. If v is specified, control is returned to the statement label associated with v in the subprogram call.

REWIND u — 6.8.1

u — is an integer variable or constant

Repositions logical unit number u to the beginning of the physical medium or to the currently opened file.

STOP [disp] — 4.8

disp — is an octal integer constant

Terminates program execution and prints the display, if one is specified.

SUBROUTINE nam[(p[,p]...)] — 8.1.3

nam — is a symbolic name

p — is a symbolic name

Begins a SUBROUTINE subprogram, indicating the program name and any dummy argument names, p.

TYPE — See WRITE, for which TYPE is a synonym.

Type declaration

typ v[,v]... — 5.2

typ — is a data type specifier, one of:

DOUBLE PRECISION
REAL
DOUBLE INTEGER
INTEGER
LOGICAL

Statement Format — Effect — Manual Section

Type Declaration (cont.)

v — is a variable name, an array name, a function or function entry name, or an array declarator.

(Type Declarations) The symbolic names, v, are assigned the specified data type in the program unit.

WRITE (u,[f][,ERR=s])[list] — 6.4.2

u — is an integer variable or constant

f — is a FORMAT statement label or an array name

s — is an executable statement label

list — is an I/O list

(Formatted Sequential WRITE) Causes one or more logical records containing the values of the variables in the optional list to be written onto device u, according to the format specification f.

WRITE (u'r,[f][,ERR=s])[list] — 6.6.2

u — is an integer variable or constant

r — is an integer expression

f — is a FORMAT statement label or an array name

s — is an executable statement label

list — is an I/O list

(Formatted Direct Access WRITE) Causes a record formed from the list and format f to be written onto record r of unit u.

WRITE (u[,ERR=s])[list] — 6.3.2

u — is an integer variable or constant

s — is an executable statement label

list — is an I/O list

(Unformatted Sequential WRITE) Causes one unformatted record containing the values of the variables in the optional list to be written onto device u.

Statement Formats	Effect	Manual Section
WRITE (u'r[,ERR=s]) [list] u is an integer variable or constant r is an integer expression s is an executable statement label list is an I/O list	(Unformatted Direct Access WRITE) Causes a record containing the values of the variables in the list to be written onto record r of logical unit u.	6.5.2
END=s,ERR=s	(Transfer of Control on end-of-file or error condition) Is an optional element in each type of I/O statement allowing the program to transfer to statement number s on an end-of-file (END=) or error (ERR=) condition.	6.7

Table B-1
FORTRAN Library 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 P	Double P
JABS(I)	Double Integer absolute value	Double I	Double I
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 precision to Real conversion	Double P	Real
DBLE(X)	Real to Double precision conversion	Real	Double P
JFIX(X)	Real to Double integer conversion	Real	Double I
JFIX(X)	Double precision to Double integer conversion	Double P	Double I
ISNGL(I)	Double integer to integer conversion	Double I	Integer
JDBLE	Integer to Double integer	Double P	Double I
JDFIX(X)	Double precision to Double integer conversion	Double P	Double I
FLOATJ(I)	Double integer to Real conversion	Double I	Real
DBLEJ(I)	Double integer to Double precision conversion	Double I	Double P
	Truncation functions return the sign of the argument * largest integer ≤ |arg|		
AINT(X)	Real to Real truncation	Real	Real
INT(X)	Real to Integer truncation	Real	Integer
IDINT(X)	Double precision to Integer truncation	Double P	Integer
JINT(X)	Real to Double integer truncation	Real	Double I
JDINT(X)	Double precision to Double integer truncation	Double P	Double I
	Remainder functions return the remainder when the first argument is divided by the second.		
AMOD(X,Y)	Real remainder (X/Y<131072)	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder (X/Y<131072)	Double P	Double P
JMOD(I,J)	Double integer remainder (I/J<131072)	Double I	Double I
	Maximum value functions return the largest value from among the argument list; ≥ 2 arguments		
AMAX0(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAX0(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double precision maximum from Double precision list	Double P	Double P
JMAX0(I,J,...)	Double integer maximum from Double integer list	Double I	Double I
	Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments		
AMIN0(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MIN0(I,J,...)	Integer minimum of Integer list	Integer	Integer
MIN1(X,Y,...)	Integer minimum of Real list	Real	Integer
DMIN1(X,Y,...)	Double minimum of Double list	Double P	Double P
JMIN0(I,J,...)	Double integer minimum of Double integer list	Double I	Double I

Table B-1 (Cont.)
FORTRAN Library Functions

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
	The transfer of sign functions return (sign of the second argument) * (absolute value of the first argument).		
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 P	Double P
JSIGN(I,J)	Double integer transfer of sign	Double I	Double I
	Positive difference functions return the first argument minus the minimum of the two arguments.		
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
JDIM(I,J)	Double integer positive difference	Double I	Double I
	Exponential functions return the value of e raised to the argument power.		
EXP(X)	e^x $(x \geq 0)$	Real	Real
DEXP(X)	e^x $(x \geq 0)$	Double P	Double P
ALOG(X)	Returns $\log_e(X)$ $(x \geq 0)$	Real	Real
ALOG10(X)	Returns $\log_{10}(X)$	Real	Real
DLOG(X)	Returns $\log_e(X)$	Double P	Double P
DLOG10(X)	Returns $\log_{10}(X)$	Double P	Double P
SQRT(X)	Square root of Real argument $(X \geq 0)$	Real	Real
DSQRT(X)	Square root of Double precision argument $(X \geq 0)$	Double P	Double P
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double P	Double P
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double P	Double P
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double P	Double P
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double P	Double P

APPENDIX C

FORTRAN PROGRAMMING EXAMPLES

Four examples of FORTRAN programs are given below. These examples are intended to show possible methods of handling Input/Output, iterative calculations, the FORTRAN Library functions, and subprogram usage in the context of problems likely to face a FORTRAN programmer. These particular programs should not be considered as the correct or optimal approach to the specified problems since many other methods are possible in each case.

The program in example one performs linear regression on a set of X,Y coordinates. The program uses standard formulae to calculate the slope and intercept of the line which best fits the data points entered. The program listing and a sample run follow:

EXAMPLE 1 LISTING:

```
        WRITE (4,5)
5       FORMAT (//' THIS PROGRAM PERFORMS LINEAR REGRESSION'/
        1 ' THE LINE WHICH BEST FITS A SET OF X,Y PAIRS IS CALCULATED'/)
10      WRITE (4,20)
20      FORMAT (/' TYPE IN THE NUMBER OF X,Y PAIRS: ')
        READ (4,50) N
50      FORMAT (I2)
        IF (N .LE. 0) STOP
        WRITE (4,60) N
60      FORMAT (' TYPE IN ',I2,' LINES OF X,Y PAIRS'/)
        SIGMXY = 0
        SIGMX = 0
        SIGMY = 0
        SIGMXX = 0
        DO 100 J=1,N
        READ (4,70) X,Y
70      FORMAT (2F8.3)
        SIGMXY = SIGMXY + X*Y
        SIGMX = SIGMX + X
        SIGMY = SIGMY + Y
```

```
100     SIGMXX = SIGMXX + X*X
        ZN = N
        A = (SIGMXY-SIGMX*SIGMY/ZN) / (SIGMXX-SIGMX*SIGMX/ZN)
        B = (SIGMY - A * SIGMX) / ZN
        WRITE (4,300) A,B
300     FORMAT (/' THE BEST FIT IS Y=',F8.3,'  X=',F8.3)
        GO TO 10
        END
```

EXAMPLE 1 SAMPLE RUN:

```
THIS PROGRAM PERFORMS LINEAR REGRESSION
THE LINE WHICH BEST FITS A SET OF X,Y PAIRS IS CALCULATED


TYPE IN THE NUMBER OF X,Y PAIRS:
10

TYPE IN 10 LINES OF X,Y PAIRS

   1.0  4.7
   2.0  9.4
   3.0  14.1
   4.0  18.8
   5.0  23.5
   6.0  28.2
  10.0  47.0
  12.0  56.4
  13.0  61.1
  20.0  94.0


THE BEST FIT IS Y=   4.700  X=   0.000

TYPE IN THE NUMBER OF X,Y PAIRS:
0

STOP  000000
```

The program in example two manipulates data representing test scores. The scores are read from the source file, placed in descending order, and sent to an output file. Then the absolute total and histogram of the test scores in each 10-point interval are output on the terminal. The program listing and a sample run follow:

EXAMPLE 2 LISTING:

```
          INTEGER STARS(80),ARRAY (200),HIST(10)
          DATA STARS/80*'*'/,IALT/#764000/
          DO 10 I=1,200
          READ (1,20,END=100) ARRAY(I)
20        FORMAT (I3)
10        CONTINUE
100       ISIZE = I-1
          II = ISIZE - 1
          DO 120 J=1,II
          KK = J+1
          DO 110 K=KK,ISIZE
          IF (ARRAY(J) .GE. ARRAY(K)) GO TO 110
          ITMP = ARRAY(J)
          ARRAY(J) = ARRAY(K)
          ARRAY(K) = ITMP
110       CONTINUE
120       CONTINUE
          DO 125 K=1,ISIZE
125       WRITE(2,20) ARRAY(K)
          DO 126 K=1,10
126       HIST(K) = 0
          DO 130 K=1,ISIZE
          N = ARRAY(K) / 10 + 1
130       HIST(N) = HIST(N) + 1
          WRITE(5,135)
135       FORMAT (1X,' THE NUMBER OF TEST SCORES AND A HISTOGRAM'/
          1 ' IN EACH 10 POINT INTERVAL FOLLOWS:'/)
          DO 150 K=10,100,10
          J = K - 10
          WRITE(5,140) HIST(K/10),J,K,IALT
140       FORMAT ('0',I3,' IN THE RANGE ',I3,' TO ',I3,A1)
          IF (HIST(K/10) .EQ. 0) GO TO 150
          JJ = HIST(K/10)
          WRITE (5,145)  (STARS(M),M=1,JJ),IALT
145       FORMAT (1H+,2X,80A1)
150       WRITE (5,146)
146       FORMAT (' ')
          WRITE(5,160) ISIZE
160       FORMAT (///' THE TOTAL NUMBER OF TEST SCORES = ',I3)
          STOP
          END
```

EXAMPLE 2 SAMPLE RUN:

```
THE NUMBER OF TEST SCORES AND A HISTOGRAM
 IN EACH 10 POINT INTERVAL FOLLOWS:

  5 IN THE RANGE   0 TO  10   *****

  0 IN THE RANGE  10 TO  20

  1 IN THE RANGE  20 TO  30   *

  2 IN THE RANGE  30 TO  40   **

  9 IN THE RANGE  40 TO  50   *********

  1 IN THE RANGE  50 TO  60   *

  6 IN THE RANGE  60 TO  70   ******

  6 IN THE RANGE  70 TO  80   ******

  3 IN THE RANGE  80 TO  90   ***

  2 IN THE RANGE  90 TO 100   **

THE TOTAL NUMBER OF TEST SCORES =   35
STOP  000000
```

Example three shows a method of calculating the prime factors of an integer. A simple table look-up method was used to determine the necessary primes. Note the unusual use of FORTRAN carriage control to facilitate the prime factor output. MOD is a Library function and is described in Section 8.2. The program listing and a sample run follow:

EXAMPLE 3 LISTING:

```
        INTEGER P,HOLD
        DATA IALT/#764000/
        WRITE(4,50)
50      FORMAT(' THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF',
        1  ' AN INTEGER < 131072.'/' ENTERING A NEGATIVE OR ZERO',
        2  ' NUMBER TERMINATES EXECUTION.'/)
80      WRITE(4,100) IALT
100     FORMAT(/' ENTER #  ',A1)
        READ(5,) NUMBER
        IF (NUMBER .LE. 0) STOP
        ISQRT = SQRT(NUMBER)
        P = 1
        IFLAG = 0
        HOLD = NUMBER
        IF (HOLD .LE. 3) GO TO 240
```

```
200     P = NPRIME(P)
205     IREM = MOD(HOLD,P)
        IF (IREM .EQ. 0) GO TO 400
        IF (P .LE. ISQRT) GO TO 200
        IF (IFLAG .NE. 0) GO TO 300
240     WRITE(4,250) NUMBER
250     FORMAT(I7,' IS A PRIME NUMBER'/)
        GO TO 80
300     IF (HOLD .GT. 1) WRITE(4,350) HOLD
350     FORMAT(I6)
        GO TO 80
400     IFLAG = 1
        HOLD = HOLD/P
        IF (HOLD .EQ. 1) GO TO 500
        WRITE(4,450) P,IALT
450     FORMAT(I6,' *',A1)
        GO TO 205
500     WRITE(4,350) P
        GO TO 80
        END
```

```
        FUNCTION NPRIME(IOLD)
        DIMENSION MPRIME(46)
        DATA MPRIME/2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,
        1   53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,
        2   137,139,149,151,157,163,167,173,179,181,191,193,197,199/
        IF (IOLD .EQ. 1) N = 0
        N = N + 1
        NPRIME = MPRIME(N)
        RETURN
        END
```

EXAMPLE 3 SAMPLE RUN:

```
THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF AN INTEGER < 131072.
ENTERING A NEGATIVE OR ZERO NUMBER TERMINATES EXECUTION.


ENTER #  2
      2 IS A PRIME NUMBER


ENTER #  16
     2 *     2 *     2 *     2

ENTER #  27
     3 *     3 *     3

ENTER #  30
     2 *     3 *     5

ENTER #  1432
     2 *     2 *     2 *   179

ENTER #  0

STOP  000000
```

APPENDIX D

ERROR MESSAGES

D.1 FORTRAN ERROR MESSAGES

FORTRAN errors are detected both at the time a program is compiled (generally syntax errors) and at the time it is executed (typically, computed values out of range or I/O device error). FORTRAN run time errors are detected and messages printed by the FORTRAN Object Time System (OTS). Other system error messages are not listed in this manual.

D.2 COMPILER ERROR MESSAGES

FORTRAN compiler error messages are printed in the form:

```
>mnA<
```

where:

mn is the error number

A is the alphabetic mnemonic characterizing the class of error

All error messages are given below.

Number	Letter	Meaning
		Common, equivalence, data errors:
01	C	No open parenthesis after variable name in DIMENSION statement
02	C	No slash after common block name
03	C	Common block name previously defined
04	C	Variable appears twice in COMMON

(continued on next page)

Error Messages

Number	Letter	Meaning
		Common, Equivalence and Data errors (cont.)
05	C	EQUIVALENCE list does not begin with open parenthesis
06	C	Only one variable in EQUIVALENCE class
07	C	EQUIVALENCE distorts COMMON
08	C	EQUIVALENCE extends COMMON down
09	C	Inconsistent EQUIVALENCing
10	C	EQUIVALENCE extends COMMON down
11	C	Illegal delimiter in EQUIVALENCE list
12	C	Non-COMMON variables in BLOCK DATA
15	C	Illegal repeat factor in DATA statement
16	C	DATA statement stores in COMMON in non-BLOCK DATA statement or in non-COMMON in BLOCK DATA statement
		DO errors:
01	D	Statement with unparenthesized = sign and comma not a DO statement
04	D	DO variable not followed by = sign
05	D	DO variable not integer
06	D	Initial value of DO variable not followed by comma
07	D	Improper delimiter in DO statement
09	D	Illegal terminating statement for DO loop
		External symbol and entry-point errors:
01	E	Variable in EXTERNAL statement not simple non-COMMON variable or simple dummy variable
02	E	ENTRY name non-unique
03	E	ENTRY statement in main program
04	E	No = sign following argument list in arithmetic statement function
05	E	No argument list in FUNCTION subprogram
06	E	Subroutine list in CALL statement already defined as variable
08	E	Function or array name used in expression without parenthesis

(continued on next page)

Error Messages

Number	Letter	Meaning
		External symbol and entry-point errors: (cont.)
09	E	Function or array name used in expression without open parenthesis
		Format errors:
01	F	Bad delimiter after FORMAT number in I/O statement
02	F	Missing field width, illegal character or unwanted repeat factor
03	F	Field width is 0
04	F	Period expected, not found
05	F	Period found, not expected
06	F	Decimal length missing (no "d" in "Fw.d")
07	F	Missing left parenthesis
08	F	Minus without number
09	F	No P after negative number
10	F	No number before P
12	F	No number or 0 before H
13	F	No number or 0 before X
15	F	Too many left parentheses
		Hollerith errors:
02	H	More than two characters in integer or logical Hollerith constant
03	H	Number preceding H not between 1 and 5
04	H	Carriage return inside Hollerith field
05	H	Number preceding H not an integer
06	H	More than five characters inside quotes
07	H	Carriage return inside quotes
		Various illegal errors:
01	I	Unidentifiable statement
02	I	Misspelled statement
03	I	Statement out of order
04	I	Executable statement in BLOCK DATA subroutine
05	I	Illegal character in I/O statement, following unit number

(continued on next page)

Error Messages

Number	Letter	Meaning
		Various illegal errors: (cont.)
06	I	Illegal delimiter in ASSIGN statement
07	I	Illegal delimiter in ASSIGN statement
08	I	Illegal type in IMPLICIT statement
09	I	Logical IF as target of logical IF
10	I	RETURN statement in main program
11	I	Semicolon in COMMON statement outside of BLOCK DATA
12	I	Illegal delimiter in IMPLICIT statement
13	I	Misspelled REAL or READ statement
14	I	Misspelled END or ENDFILE statement
15	I	Misspelled ENDFILE statement
16	I	Statement function out of order or undimensioned array
17	I	Typed FUNCTION statement out of order
18	I	Illegal character in context
19	I	Illegal logical or relational operator
20	I	Illegal letter in IMPLICIT statement
21	I	Illegal letter range in IMPLICIT statement
22	I	Illegal delimiter in letter section of IMPLICIT statement
23	I	Illegal character in context
24	I	Illegal comma in GOTO statement
26	I	Illegal variable used in multiple RETURN statement
		Pushdown list errors:
01	L	DO nesting too deep
02	L	Illegal DO nesting
03	L	Subscript/function nesting too deep
04	L	Incomplete DO loop caused by backwards DO loop or error in DO loop foot statement or I/O statement with implied DO loop

(continued on next page)

Error Messages

Number	Letter	Meaning
		Overflow errors:
01	M	EQUIVALENCE class list full
02	M	Program size exceeds 8K
03	M	Local array length larger than 8K
04	M	Element position in local array larger than 8K or in common array larger than 32K (EQUIVALENCE, DATA)
06	M	Integer negative or larger than 131071
07	M	Exponent of floating point number larger than 76
08	M	Overflow accumulating constant - too many digits
09	M	Overflow accumulating constant - too many digits
10	M	Overflow accumulating constant - too many digits
		Statement number errors:
01	N	Multiply defined statement number or compiler error
02	N	Statement erroneously labeled
03	N	Undefined statement number
04	N	FORMAT statement without statement number
05	N	Statement number expected, not found
07	N	Statement number more than five digits
08	N	Illegal statement number
09	N	Invalid statement label or continuation
		Partword errors:
01	P	Expected colon, found none
02	P	Expected close bracket, found none
03	P	Last bit number larger than 35
04	P	First bit number larger than last bit number.
05	P	First and last bit numbers not simple integer constants

(continued on next page)

Error Messages

Number	Letter	Meaning
		Subscripting errors:
01	S	Illegal subscript delimiter in specification statements
02	S	More than three subscripts specified
03	S	Illegal delimiter in subroutine argument list
04	S	Non-integer subscript
05	S	Non-scalar subscript
06	S	Integer scalar expected, not found
10	S	Two operators in a row
11	S	Close parenthesis following an operator
12	S	Adjustable dimension not in dummy array
13	S	Adjustable dimension not a dummy integer
14	S	Two arguments in a row
15	S	Digit or letter encountered after argument conversion
16	S	Number of subscripts stated not equal to number declared
		Table overflow errors:
01	T	Arithmetic statement, computed GOTO list, or DATA statement list too large
02	T	Too many dummy variables in arithmetic statement function
03	T	Symbol and constant tables overlap
		Variable errors:
01	V	Two modes specified for same variable name
02	V	Variable expected, not found
03	V	Constant expected, not found
04	V	Array defined twice
05	V	Error: variable is EXTERNAL or argument (EQUIVALENCE, DATA)
07	V	More than one dimension indicated for scalar variable
08	V	First character after READ or WRITE not open parenthesis in I/O statement

(continued on next page)

Error Messages

Number	Letter	Meaning
		Variable errors: (cont.)
09	V	Illegal constant in DATA statement
11	V	Variables outnumber constants in DATA statement
12	V	Constants outnumber variables in DATA statement
14	V	Illegal dummy variable (previously used as non-dummy variable)
16	V	Logical operator has non-integer, non-logical arguments
17	V	Illegal mixed mode expression
19	V	Logical operator has non-integer, non-logical arguments or unbalanced parentheses
21	V	Signed variable left of equal sign
22	V	Illegal combination for exponentiation
25	V	.NOT. operator has non-integer, non-logical argument
27	V	Function in specification statement
28	V	Two exponents in one constant
29	V	Illegal redefinition of a scalar as a function
30	V	No number after E or D in a constant
32	V	Non-integer record number in random access I/O
35	V	Illegal delimiter in I/O statement
36	V	Illegal syntax in READ, WRITE, ENCODE, or DECODE statement
37	V	END and ERR exits out of order in I/O statement
38	V	Constant and variable modes don't match in DATA statement
39	V	ENCODE or DECODE not followed by open parenthesis
40	V	Illegal delimiter in ENCODE/DECODE statement
41	V	Array expected as first argument of ENCODE/ DECODE statement
42	V	Illegal delimiter in ENCODE/DECODE statement

(continued on next page)

Error Messages

Number	Letter	Meaning
		Expression errors:
01	X	Carriage return expected, not found
02	X	Binary WRITE statement with no I/O list
03	X	Illegal element in I/O list
04	X	Illegal statement number list in computed or assigned GOTO
05	X	Illegal delimiter in computed GOTO
07	X	Illegal computed GOTO statement
10	X	Illegal delimiter in DATA statement
11	X	No close parenthesis in IF statement
12	X	Illegal delimiter in arithmetic IF statement
13	X	Illegal delimiter in arithmetic IF statement
14	X	Expression on left of equals sign in arithmetic statement
15	X	Too many right parentheses
16	X	Illegal open parenthesis (in specification statements)
17	X	Illegal open parenthesis
19	X	Too many right parentheses
20	X	Illegal alphabetic in numeric constant
21	X	Symbol contains more than six characters
22	X	.TRUE., .FALSE., or .NOT. preceded by an argument
23	X	Unparenthesized comma in arithmetic expression
24	X	Unary minus in I/O list
26	X	Illegal delimiter in I/O list
27	X	Unterminated implied - DO loop in I/O list
28	X	Illegal equals sign in I/O list
29	X	Illegal partword operator
30	X	Illegal arithmetic expression
31	X	Illegal operator sequence
32	X	Illegal use of =

(continued on next page)

Number	Letter	Meaning
		Expression errors: (cont.)
33	X	Missing parentheses in I/O statement with implied DO loop - will also cause >04L<
34	X	Extraneous characters within or at end of expression

D.3 OTS ERROR MESSAGES

Following is a list of OTS error messages. (R) indicates a recoverable error; (T) a terminal error.

FORTRAN OTS error messages in XVM/DOS are printed on the console terminal (or line printer in BOSS XVM) in the form

```
.OTS nn
PC=mmmmmm
```

where

nn is an octal error number. The recoverable/non-recoverable indicator in the table below is not printed as part of the message.

mmmmmm is the 6 octal digit program counter where the error occurred

FORTRAN OTS error messages in XVM/RSX are printed on the device associated with LUN-3 in the following form:

```
OTS = nn-tasknm
PC = mmmmmm
```

where

nn is a 2-digit octal error code

tasknm is the name of the task with the error

mmmmmm is the 6-octal digit program counter where the error occurred. (Relative to the start of the partition if normal mode task.

Error Messages

Error Number		Error Description
	05 (R)	Negative REAL square root argument
	06 (R)	Negative DOUBLE PRECISION square root argument
	07 (R)	Illegal index in computed GO TO
	10 (T)	Illegal I/O device number
	11 (T)	Bad input data - IOPS mode incorrect
	12 (T)	Bad FORMAT
	13 (T)	Negative or zero REAL logarithmic argument
	14 (R)	Negative or zero DOUBLE PRECISION logarithmic argument
	15 (R)	Zero raised to a zero or negative power (zero result is passed)
	16 (R)	ATAN2 (0.0.0.0) attempted; PI/2 returned
	17 (R)	DATAN2 (0.0,D0,0.0D0) attempted; PI/2 returned
	20 (T)	Fatal I/O error (RSX only)
direct access errors	21 (T)	Undefined file
	22 (T)	Illegal record size
	23 (T)	Size discrepancy
	24 (T)	Too many records per file or illegal record number
	25 (T)	Mode discrepancy
	26 (T)	Too many open files
	30 (R)	Single integer overflow[1]
	[2]31 (R)	Extended (double) integer overflow[4]
	[2]32 (R)	Single floating point overflow
	[2]33 (R)	Double floating point overflow†
	[2]34 (R)	Single floating point underflow
	[2]35 (R)	Double floating point underflow†
	[2]36 (R)	Floating point divide check

[1] Only detected when fixing a floating point number.

[2] Also prints out PC with FPP system.

[3] If extended integer divide check, prints out PC with FPP system.

[4] With non-floating Point Processor system, only detected when fixing a floating point number.

†Not detected by software floating point routines (only by FPP system).

Error Messages

Error Number	Error Description
[1]37 (R)	Integer divide check
40 (T)	Illegal number of characters specified [legal: 0<c≤626]
41 (R)	Array exceeded
42 (T)	Bad input data
[2]50 (T)	FPP memory protect/non-existent memory
51 (T)	READ to WRITE illegal I/O Direction Change to Disk without intervening CLOSE or REWIND
52 (T)	Attempt to initialize JEA register on machine without floating point hardware.

The arithmetic errors which result in the printing of the error messages .OTS 3Ø through 37 are all recoverable errors, which is to say that program execution continues after the message has been printed. In most cases, as the table below indicates, a value is assumed as the final result of the computation. Where a "none" value is indicated, the results are meaningless. Results differ depending upon whether or not floating point hardware is used.

Error	ASSUMED VALUE	
	FPP Hardware	FPP Software
Single Floating Overflow (.OTS 32)	± largest single floating value	± largest single floating value
Double Floating Overflow (.OTS 33)	± largest single floating value	not detected
Single Floating Underflow (.OTS 34)	zero	zero
Double Floating Underflow (.OTS 35)	zero	not detected
Floating Divide Check (.OTS 36)	± largest single floating value	± largest single floating value
Integer Overflow (.OTS 30)	limited detection[3]	limited detection[3]

[1] If extended integer divide check, prints out PC with FPP system.
[2] Also prints out PC with FPP system.
[3] When fixing a floating point number, integer and double integer overflow is detected. In these instances, plus or minus the largest integer for the data mode is assumed as the result.

Error Messages

	ASSUMED VALUE	
Error	FPP Hardware	FPP Software
Double Integer Overflow (.OTS 31)	none[2]	limited detection[1]
Integer Divide Check (.OTS 37)	none	none

[1] When fixing a floating point number, integer and double integer overflow is detected. In these instances, plus or minus the largest integer for the data mode is assumed as the result.

[2] With the FPP hardware all extended (double) integer overflow conditions are detected, but the results are meaningless. Further, when converting a double integer, the magnitude of which is $>2^{17}-1$, to a single integer, no error is indicated and the high order digits are lost.

APPENDIX E

EXTENSIONS AND RESTRICTIONS TO ANSI 1966 STANDARD

FORTRAN for the XVM conforms to the specifications for American National Standard FORTRAN X3.9-1966 with extensions and with restrictions. Numbers in parentheses refer to applicable sections of the standard.

E.1 EXTENSIONS TO STANDARD FORTRAN

1. Alphanumeric Literals - Alphanumeric literals, character strings bounded by apostrophes, may be used in place of Hollerith constants. (5.1.1.6)

2. Array Subscripts - Subscript expressions for arrays may be any valid integer-type expression provided any array elements contained in the expressions have only a single subscript. (5.1.3.3)

3. ASSIGN Statement - An integer variable assigned the value of a statement label in an ASSIGN statement may be used as a statement label argument in an arithmetic IF statement, a CALL or function reference as well as an assigned GO TO statement. (7.1.1.3 and 7.1.2.2)

4. Assigned GO TO Statement - The statement label list in an assigned GO TO statement is optional. (7.1.2.1.2)

5. Assignment Statements - Multiple variables may have values assigned to them in a single arithmetic or logical assignment statement, as in A=B=C=1.

6. CALL Statements - The arguments to a subroutine referenced in a CALL statement may include statement labels, each preceded by the @ sign, to facilitate return from the subroutine to statements other than the one following the CALL.

7. COMMON Statements - An extension of the form of the COMMON statements permits data initialization specifications, which conform to those for DATA statements, to appear directly in a COMMON statement which is part of a BLOCK DATA subprogram.

8. DATA Statements - An implied DO loop, similar to that used in an I/O statement, may be used in a DATA statement.

9. Direct Access Input/Output - Direct access files, both formatted and unformatted, are supported on disk. The CALL DEFINE statement creates and opens the file for access. Data are transferred to and from records in any random order of record reference by use of extended forms of the READ and WRITE statements.

10. DO Statement - The DO statement may have a negative increment as well as negative or zero initial and terminal values. (7.1.2.8)

11. DOUBLE INTEGER Type - A double word integer data type is supported extending the range for integer data.

12. ENCODE/DECODE Statements - The ENCODE and DECODE statements implement memory-to-memory data conversions from/to ASCII to/from binary form under format control.

13. END=s/ERR=s Input/Output Option - The specifications END=s and ERR=s, where s represents a statement number, may be included in any READ or WRITE statement to transfer control to the specified statements upon detection of an end-of-file, end-of-medium or an error condition.

14. ENTRY Statements - ENTRY statements may be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single routine.

15. FORMAT Specification - Additional format conversion specifiers have been defined: O (octal conversion), R (alphanumeric conversion, right-adjusted), and alphanumeric literals (quoted strings which replace nH Holerith fields). Also, the T specification permits field alignment with numbered columns.

16. FUNCTION Statements - The arguments to a FUNCTION subprogram passed in a function reference may include statement labels, each preceded by the @ sign to facilitate error returns to statements other than the one in which the function reference appears.

17. IMPLICIT Statement - The IMPLICIT statement has been added to permit the user to redefine the implied data type of symbolic names.

18. Implied-Format Input/Output - In a modified form of formatted READ and WRITE statements where no reference is made to format statements or arrays, data conversion is governed by an implied format which is directed by the data type of each element in the input/output list.

19. Input/Output Extensions - Several subroutines have been made available to perform special purpose input/output functions, notably those to access named files on directoried devices.

20. Mixed-Mode Expressions - Mixed mode expressions may contain elements of any data type (6.1).

21. Octal Constants - Numeric quantities may be specified in octal form as well as in decimal.

22. Part-Word Notation - Elements in arithmetic expressions may be modified by using a "part-word" notation which permits specification of bit fields to be used.

23. RETURN Statement - The RETURN statement may reference a dummy argument which corresponds to a statement number in the actual argument list to facilitate a conditional return to a statement other than the one following the CALL or function reference.

24. WRITE Statement - Two synonyms are permitted for the WRITE statement: PRINT and TYPE.

25. .XOR. Operator - An additonal logical operator, .XOR., is provided to perform the "exclusive or" function. (6.3)

E.2 RESTRICTIONS TO STANDARD FORTRAN

1. BLOCKDATA Subprogram - Within a BLOCKDATA subprogram several common blocks may be defined but only the last one may be initialized with data. (8.5)

2. COMMON Statement - The name of a named common block must be unique to the program unit in which it appears. (7.2.13 and 10.1.1)

3. COMPLEX Type - The COMPLEX data type is not supported. (4.2.4, 5.1.1.4, 7.2.1.6 and 8.2)

4. EQUIVALENCE Statement - The EQUIVALENCE statement may not produce redundant or circular definitions. (7.2.1.4)

5. Hollerith Constants - Hollerith data does not have a unique data type. Instead, it is typed as either REAL or as DOUBLE INTEGER, depending upon the form used. (4.2.6)

6. Specification Statements - The specification statements must appear in a definite order.

INDEX

FORTRAN IV XVM
Language Manual
DEC-XV-LF4MA-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Repcrt (SPR) form.

Did you find errors in this manual? If so, specify by page.

____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____

Please indicate the type of user/reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Non-programmer interested in computer concepts and capabilities

Name ____________________ Date __________

Organization ______________________________

Street ______________________________

City __________ State __________ Zip Code or Country __________

If you require a written reply, please check here. □

Please cut along this line.

-- Fold Here --

-- Do Not Tear - Fold Here and Staple --

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

PRINTED IN U.S.A.