PDP-11

FORTRAN

Language Reference Manual

DEC-11-LFLRA-B-D

pdp11

digital

PDP-11

FORTRAN

Language Reference Manual

DEC-11-LFLRA-B-D

Order additional copies as directed on the Software Information page at the back of this document.

digital equipment corporation · maynard, massachusetts

First Printing, June 1974

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 manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

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

Copyright © 1974, Digital Equipment Corporation

The HOW TO OBTAIN SOFTWARE INFORMATION page, located at the back of this document, explains the various services available to DIGITAL software users.

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:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

CONTENTS

			Page
CHAPTER	1	INTRODUCTION TO PDP-11 FORTRAN	1-1
	1.1	LANGUAGE OVERVIEW	1-1
	1.2	ELEMENTS OF A FORTRAN PROGRAM	1-3
	1.2.1	Statements	1-3
	1.2.2	Comments	1-4
	1.2.3	The FORTRAN Character Set	1-4
	1.3	FORMATTING A FORTRAN LINE	1-5
	1.3.1	Using FORTRAN Coding Forms	1-5
	1.3.2	Using a Text Editor	1-6
	1.3.3	Statement Number Field	1-7
	1.3.3.1	Comment Indicator	1-7
	1.3.3.2	Debug Statement Indicator	1-7
	1.3.4	Continuation Field	1-8
	1.3.5	Statement Field	1-8
	1.3.6	Sequence Number Field	1-8
	1.4	PROGRAM UNIT STRUCTURE	1-9
CHAPTER	2	FORTRAN STATEMENT COMPONENTS	2-1
	2.0	SYMBOLIC NAMES	2-2
	2.1	DATA TYPES	2-2
	2.2	CONSTANTS	2-5
	2.2.1	Integer Constants	2-5
	2.2.2	Real Constants	2-6
	2.2.3	Double Precision Constants	2-7
	2.2.4	Complex Constants	2-8
	2.2.5	Octal Constants	2-8
	2.2.6	Logical Constants	2-9
	2.2.7	Hollerith Constants	2-9
	2.2.7.1	Alphanumeric Literals	2-10
	2.2.8	Radix-50 Constants	2-10
	2.3	VARIABLES	2-11
	2.3.1	Data Type by Definition	2-12
	2.3.2	Data Type by Implication	2-12
	2.3.3	Assigning Hollerith Data to Variables	2-12
	2.3.4	LOGICAL*1 Variables	2-13

			Page
	2.4	ARRAYS	2-13
	2.4.1	Array Declarators	2-14
	2.4.2	Array Storage	2-15
	2.4.3	Data Type of an Array	2-16
	2.4.4	Subscripts	2-16
	2.5	EXPRESSIONS	2-17
	2.5.1	Arithmetic Expressions	2-17
	2.5.1.1	Use of Parentheses	2-19
	2.5.1.2	Data Type of an Arithmetic Expression	2-20
	2.5.2	Relational Expressions	2-22
	2.5.3	Logical Expressions	2-23
CHAPTER	3	ASSIGNMENT STATEMENTS	3-1
	3.1	ARITHMETIC ASSIGNMENT STATEMENT	3-1
	3.2	LOGICAL ASSIGNMENT STATEMENT	3-4
	3.3	ASSIGN STATEMENT	3-5
CHAPTER	4	CONTROL STATEMENTS	4-1
	4.1	GO TO STATEMENTS	4-1
	4.1.1	Unconditional GO TO Statement	4-2
	4.1.2	Computed GO TO Statement	4-2
	4.1.3	Assigned GO TO Statement	4-3
	4.2	IF STATEMENTS	4-4
	4.2.1	Arithmetic IF Statement	4-4
	4.2.2	Logical IF Statement	4-5
	4.3	DO STATEMENT	4-6
	4.3.1	Nested DO Loops	4-8
	4.3.2	Control Transfers in DO Loops	4-9
	4.3.3	Extended Range	4-9
	4.4	CONTINUE STATEMENT	4-11
	4.5	CALL STATEMENT	4-11
	4.6	RETURN STATEMENT	4-12
	4.7	PAUSE STATEMENT	4-12
	4.8	STOP STATEMENT	4-13
	4.9	END STATEMENT	4-13

			Page
CHAPTER	5	INPUT/OUTPUT STATEMENTS	5-1
	5.1	OVERVIEW	5-1
	5.1.1	Input/Output Devices	5-2
	5.1.2	Format Specifiers	5-3
	5.1.3	Input/Output Records	5-3
	5.2	INPUT/OUTPUT LISTS	5-3
	5.2.1	Simple Lists	5-3
	5.2.2	Implied DO Lists	5-4
	5.3	UNFORMATTED SEQUENTIAL INPUT/OUTPUT	5-6
	5.3.1	Unformatted Sequential READ Statement	5-6
	5.3.2	Unformatted Sequential WRITE Statement	5-7
	5.4	FORMATTED SEQUENTIAL INPUT/OUTPUT	5-8
	5.4.1	Formatted Sequential READ Statement	5-8
	5.4.2	Formatted Sequential WRITE Statement	5-9
	5.4.3	ACCEPT Statement	5-10
	5.4.4	TYPE Statement	5-11
	5.4.5	PRINT Statement	5-11
	5.5	UNFORMATTED DIRECT ACCESS INPUT/OUTPUT	5-12
	5.5.1	Unformatted Direct Access READ Statement	5-12
	5.5.2	Unformatted Direct Access WRITE Statement	5-13
	5.5.3	DEFINE FILE Statement	5-14
	5.6	FORMATTED DIRECT ACCESS INPUT/OUTPUT	5-15
	5.6.1	Formatted Direct Access READ Statement	5-15
	5.6.2	Formatted Direct Access WRITE Statement	5-16
	5.7	TRANSFER OF CONTROL ON END-OF-FILE OR ERROR CONDITIONS	5-16
	5.8	AUXILIARY INPUT/OUTPUT STATEMENTS	5-18
	5.8.1	REWIND Statement	5-18
	5.8.2	BACKSPACE Statement	5-18
	5.8.3	ENDFILE Statement	5-19
	5.8.4	FIND Statement	5-19
	5.8.5	OPEN Statement	5-20
	5.8.5.1	UNIT Keyword	5-22
	5.8.5.2	NAME Keyword	5-22
	5.8.5.3	TYPE Keyword	5-22
	5.8.5.4	ACCESS Keyword	5-23
	5.8.5.5	READONLY Keyword	5-23
	5.8.5.6	FORM Keyword	5-23
	5.8.5.7	RECORDSIZE Keyword	5-24
	5.8.5.8	ERR Keyword	5-24
	5.8.5.9	BUFFERCOUNT Keyword	5-24
	5.8.5.10	INITIALSIZE Keyword	5-25
	5.8.5.11	EXTENDSIZE Keyword	5-25
	5.8.5.12	NOSPANBLOCKS Keyword	5-25
	5.8.5.13	SHARED Keyword	5-25
	5.8.5.14	DISPOSE Keyword	5-26
	5.8.5.15	ASSOCIATEVARIABLE Keyword	5-26
	5.8.5.16	CARRIAGECONTROL Keyword	5-26
	5.8.5.17	MAXREC Keyword	5-26

			Page
	5.8.6	CLOSE Statement	5-27
	5.9	ENCODE AND DECODE STATEMENTS	5-27
CHAPTER	6	FORMAT STATEMENTS	6-1
	6.1	OVERVIEW	6-1
	6.2	FIELD DESCRIPTORS	6-2
	6.2.1	I Field Descriptor	6-2
	6.2.2	O Field Descriptor	6-3
	6.2.3	F Field Descriptor	6-4
	6.2.4	E Field Descriptor	6-5
	6.2.5	D Field Descriptor	6-6
	6.2.6	G Field Descriptor	6-7
	6.2.7	L Field Descriptor	6-8
	6.2.8	A Field Descriptor	6-9
	6.2.9	H Field Descriptor	6-10
	6.2.9.1	Alphanumeric Literals	6-11
	6.2.10	X Field Descriptor	6-11
	6.2.11	T Field Descriptor	6-12
	6.2.12	Q Field Descriptor	6-12
	6.2.13	$ Descriptor	6-13
	6.2.14	Complex I/O	6-13
	6.2.15	Scale Factor	6-14
	6.2.16	Grouping and Group Repeat Specifications	6-15
	6.2.17	Variable Format Expressions	6-16
	6.3	CARRIAGE CONTROL	6-17
	6.4	FORMAT SPECIFICATION SEPARATORS	6-18
	6.5	EXTERNAL FIELD SEPARATORS	6-18
	6.6	OBJECT TIME FORMAT	6-19
	6.7	FORMAT CONTROL INTERACTION WITH INPUT/OUTPUT LISTS	6-20
	6.8	SUMMARY OF RULES FOR FORMAT STATEMENTS	6-21
	6.8.1	General	6-21
	6.8.2	Input	6-22
	6.8.3	Output	6-23
CHAPTER	7	SPECIFICATION STATEMENTS	7-1
	7.1	IMPLICIT STATEMENT	7-1
	7.2	TYPE DECLARATION STATEMENTS	7-2
	7.3	DIMENSION STATEMENT	7-4
	7.3.1	Adjustable Dimensions	7-5

		Page
7.4	COMMON STATEMENT	7-6
7.4.1	Blank Common and Named Common	7-6
7.4.2	COMMON Statements with Array Declarators	7-8
7.5	EQUIVALENCE STATEMENT	7-8
7.5.1	Making Arrays Equivalent	7-9
7.5.2	EQUIVALENCE and COMMON Interaction	7-11
7.5.3	EQUIVALENCE and LOGICAL*1 Arrays	7-11
7.6	EXTERNAL STATEMENT	7-12
7.7	DATA STATEMENT	7-13
7.8	PARAMETER STATEMENT	7-14
7.9	PROGRAM STATEMENT	7-15
CHAPTER 8	SUBPROGRAMS	8-1
8.1	USER-WRITTEN SUBPROGRAMS	8-1
8.1.1	Arithmetic Statement Function (ASF)	8-1
8.1.2	FUNCTION Subprogram	8-3
8.1.3	SUBROUTINE Subprogram	8-5
8.1.4	ENTRY Statement	8-7
8.1.4.1	ENTRY in Function Subprograms	8-8
8.1.4.2	ENTRY and Array Declarator Interaction	8-8
8.1.5	BLOCK DATA Subprogram	8-10
8.2	FORTRAN LIBRARY FUNCTIONS	8-11
8.2.1	Generic Function References	8-14
8.2.2	Processor-Defined Function References	8-15
8.2.3	Generic and Processor-Defined Function Usage	8-16
APPENDIX A	CHARACTER CODES	A-1
A.1	FORTRAN CHARACTER SET	A-1
A.2	ASCII CHARACTER SET	A-2
A.3	RADIX-50 CHARACTER SET	A-3
APPENDIX B	FORTRAN LANGUAGE SUMMARY	B-1
B.1	EXPRESSION OPERATORS	B-1
B.2	STATEMENTS	B-2
B.3	LIBRARY FUNCTIONS	B-17
APPENDIX C	FORTRAN PROGRAMMING EXAMPLES	C-1

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.

Since this document serves as the FORTRAN Language Reference manual for several of the operating systems which run on the PDP-11 Family of computers, it makes no reference to system dependent information. Associated with this document, however, should be the FORTRAN User's Guide containing the necessary information for running a FORTRAN program on a specific operating system.

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)
△	(delta)	Space character (SPACE bar)

FORTRAN IV-PLUS is a superset of FORTRAN IV. Language elements that are common to both processors are presented without background shading. Language elements that are available only in FORTRAN IV-PLUS are printed against a shaded background.

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 PDP-11 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. PDP-11 FORTRAN conforms to the specifications for American National Standard FORTRAN X3.9-1966. The following enhancements to American National Standard FORTRAN are included in PDP-11 FORTRAN:

1. Any arithmetic expression may be used as an array subscript. If the value of the expression is not an integer, it is converted to Integer format.

2. Character literals (character strings bounded by apostrophes) may be used in place of Hollerith constants.

3. Mixed-mode expressions may contain elements of any data type, including complex.

4. The statement label list in an assigned GO TO statement is optional.

5. The following Input/Output statements have been added:

Statements	
ACCEPT TYPE PRINT	Device-oriented I/O
DEFINE FILE READ (u'r) WRITE (u'r) FIND (u'r)	Unformatted Direct Access I/O

(The letter "u" represents a logical unit number and the letter "r" represents a record number.)

6. The specifications END=n and/or ERR=n (where "n" represents a statement number) may be included in any READ or WRITE statement to transfer control to the specified statement upon detection of an end-of-file or error condition.

7. The following additional data type is provided:

   LOGICAL*1

8. The IMPLICIT statement has been added to permit the user to redefine the implied data type of symbolic names.

9. Any FORTRAN statement may be followed, in the same line, by an explanatory comment that begins with an exclamation point.

10. Statements that are included in a program for debugging purposes may be so designated by the letter D in column 1. Those statements are compiled only when the associated compiler option switch is set; they are treated as comments otherwise.

11. Undersized input data fields may contain external field separators to override the FORMAT field width specifications for those fields (called "short field termination").

12. General expressions are permitted for the initial value, increment, and limit parameters in the DO statement, and as the control parameter in the computed GO TO statement.

13. The value of the DO statement increment parameter may be negative.

14. General expressions are permitted in I/O lists of WRITE, TYPE, and PRINT statements.

In addition, FORTRAN IV-PLUS includes the following extensions:

15. A PROGRAM statement may be used in a main program.

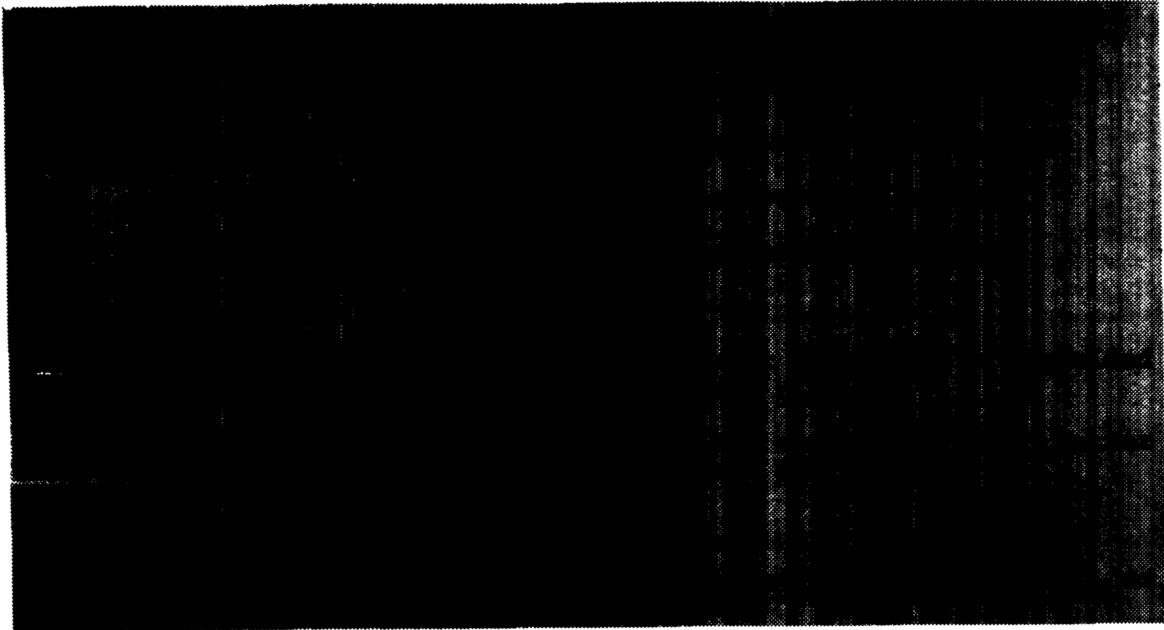
16. ENTRY statements may be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single routine.

17. PARAMETER statements may be used to give symbolic names to constants.

18. Lower bounds of the array dimension may be specified in all array declarators.

19. The data type INTEGER*4 provides a sign plus 31 bits of precision.

20. A compiler command specification allows all INTEGER and LOGICAL declarations (without explicit length specification) to be considered as either INTEGER*2 and LOGICAL*2, or INTEGER*4 and LOGICAL*4, respectively.



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). One or more program units (one main program and possibly one or more subprograms) comprise the executable program.

1.2.1 Statements

Statements are grouped into two general classes: executable and nonexecutable. Executable statements describe the action of the program; nonexecutable statements describe data arrangement and characteristics, and provide editing and data conversion information.

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.)

A statement may be identified by a statement label so that other statements can refer to it, either for the information it contains or to transfer control to it. A statement label has the form of an integer number placed in the first five columns of a statement's initial line.

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. Also, if an exclamation point (!) is placed in the statement portion of a source line, the rest of that line is treated 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
	Space or blank or tab
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe

"	Double Quote
$	Dollar Sign

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

1.3 FORMATTING A FORTRAN LINE

The formatting of a FORTRAN line is 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

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, 2
      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
      TYPE 105, I
10    CONTINUE

105   FORMAT (I4,' IS PRIME.')
      END
```

PG-3 DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS

Figure 1-1
FORTRAN Coding Form

1.3.2 Using a Text Editor

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 or a D in column 1, and the following character is a 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.

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)=111200022D0          or      ΔΔΔΔΔΔSEL(1)=111200022D0
```

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 alphanumeric literal (for example, GO TO and GOTO are equivalent). The TAB character in a statement field is treated the same as a space by the compiler. In the source listing produced by the compiler, the TAB causes the character that follows to be printed at the next tab stop (located at columns 9,17,25,33, etc.).

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.

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, then ignores the line.

1.3.3.2 Debug Statement Indicator - Debug statements are designated by a D in column 1. The initial line of the debug statement may contain a statement label in columns 2-5. If a debug statement is continued onto more than one line, then every continuation line must contain a D in column 1 as well as a continuation character in column 6.

The debug statement can be treated either as source text to be compiled or as a comment, depending on the setting of a compiler command switch. When the switch is set, debug statements are compiled as a part of the source program; when it is not set, debug statements are treated as comments.

1.3.4 Continuation Field

Column 6 of a FORTRAN line is reserved for a continuation indicator. Any character except zero or space in this column is recognized as a continuation indicator. A common practice 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.

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.

Comment lines cannot be continued. All comment lines must begin with the letter C in column 1. Comment lines must not intervene between a statement's initial line and its continuation line(s), or between successive continuation lines.

1.3.5 Statement Field

The text of a FORTRAN statement is placed in columns 7 through 72. Because the compiler ignores the TAB character and spaces (except in Hollerith constants and alphanumeric literals), 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 PARAMETER statements.

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or
BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
Statements</td><td rowspan="2">PARAMETER
Statements</td><td>IMPLICIT
Statements</td></tr>
<tr><td>Other Specification
Statements</td></tr>
<tr><td rowspan="2">DATA
Statements</td><td>Statement
Function
Definitions</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Line</td></tr>
</table>

Figure 1-2
Required Order of Statements and Lines

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-describing 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.

4. An expression may be a single constant, variable, array element reference, or function reference, or it may be a combination of those components and certain other elements, called operators, that specify computations to be performed on the values represented by those components to obtain a single result.

5. 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.

2.0 SYMBOLIC NAMES

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

A symbolic name is a string of letters and digits, the first of which must be a letter. The name may be of any length, but characters after the sixth are ignored. Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(Begins with a numeral)
K9	B.4	(Contains a special character)
X		

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
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Within one program unit, the same symbolic name cannot be used to identify more than one entity, except as noted. Within an executable program, the same symbolic name can be used to identify only one of the entities indicated as "Unique in Executable Program".

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 7.1 and 7.2.

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

2.1 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, 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, a decimal fraction, or a combination of the two
3. Double Precision - Similar to real, but with more than twice the degree of accuracy in its representation
4. Complex - A pair of real values that represents a complex number; the first represents the real part of that number, the second represents the imaginary part
5. Logical - The logical value "true" or "false"

An important attribute of each type of data is the amount of computer memory required to represent a value of that type. Variations on the basic types affect either the accuracy of the represented value or the allowed range of values.

Standard FORTRAN specifies that a "storage unit" is the amount of storage needed to represent a Real, Integer or Logical Value. Double Precision and Complex values occupy two storage units. In PDP-11 FORTRAN a storage unit corresponds to four bytes (two words) of memory.

PDP-11 FORTRAN provides additional types of data for optimum selection of performance and memory requirements.

Table 2-1
Data Type Storage Requirements

DATA TYPE	FORTRAN IV (Bytes)
INTEGER	2 or 4 (Note 1)
INTEGER*2	2
INTEGER*4	4 (Note 3)
REAL	4
DOUBLE PRECISION	8
COMPLEX	8
BYTE	1 (Note 6)
LOGICAL	4
LOGICAL*1	1
LOGICAL*2	Not Available
LOGICAL*4	4

NOTES:

1. Either two or four bytes are allocated according to a compiler command specification. Two bytes is the default allocation. In either case only two bytes are used to represent the integer value. (The 4-byte allocation is provided to simplify use of programs developed on other FORTRAN systems. Consult the FORTRAN IV User's Guide for details.)

2. Either two or four bytes are allocated depending on a compiler command specification. Two bytes is the normal (default) allocation. When four bytes are allocated, all four bytes are used to represent the integer value. Hence 4-byte integers may be used to store a larger range of values than 2-byte integers. (4-byte integers also sometimes simplify the use of programs developed on other FORTRAN systems.)

3. Four bytes are allocated but only the first two are used to represent the integer value. The range of possible values is therefore the same as for INTEGER*2 variables.

4. Either two or four bytes are allocated depending on a compiler command specification. Two bytes is the normal case.

5. LOGICAL*1 variables are also referred to as BYTE variables.

6. The keyword BYTE is not available. However, LOGICAL*1 is equivalent.

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

2.2 CONSTANTS

A constant represents a fixed value. A constant can represent a numeric value, a logical value, or a character string.

2.2.1 Integer Constants

An integer constant is a whole number with no decimal point. It may have a leading sign, and is interpreted as a decimal number.

The general form for an integer constant is:

[±]nn

where nn is a string of numeric characters. Leading zeros, if any, are ignored.

A negative 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, an integer constant cannot contain any character other than the numerals 0 through 9.

The absolute magnitude of an integer constant cannot be greater than 32767 in FORTRAN IV or 2147483647 in FORTRAN IV-PLUS.

Examples

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

In FORTRAN IV-PLUS, if the value of the [illegible] range -32768 to +32767 then it [illegible] and is treated as INTEGER*2 data type. [illegible] then it [illegible] a 32-bit [illegible] as INTEGER*4 data type.

2.2.2 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. The number of digits is not limited, but only the leftmost eight digits are significant. Leading zeros (zeros to the left of the first non-zero digit) are ignored when counting the leftmost eight digits. Thus in the constant 0.000012345678 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 constant or an integer constant followed by a decimal exponent of the form:

E[±]nn

where nn is a 1- or 2-digit 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 1.0×10^6).

A real constant occupies two words of PDP-11 storage and is interpreted as a real number having a degree of precision slightly greater than seven decimal digits.

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 1.7×10^{38}, nor smaller in magnitude than 0.29×10^{-38}.

Examples

Valid Real Constants	Invalid Real Constants	
3.14159	1,234,567	(Commas not allowed)
621712.	325E-75	(Too small)
-.00127	-47.E47	(Too large)
+5.0E3	100	(Decimal point missing)
2E-3	$25.00	(Special character not allowed)

2.2.3 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 integer constant. The number of digits that precede the exponent is not limited, but only the leftmost 17 digits are significant.

A double precision constant occupies four words of PDP-11 storage and is interpreted as a real number having a degree of precision approximately equal to 17 significant digits.

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 0.29×10^{-38}, nor greater than 1.7×10^{38}.

Examples

```
1234567890D+5
+2.71828182846182D00
-72.5D-15
  1D0
```

2.2.4 Complex Constants

A complex constant is a pair of signed or unsigned real constants separated by a comma and enclosed in parentheses. The first real constant represents the real part of that number and the second represents the imaginary part.

A complex constant takes the form:

```
(rc,rc)
```

where "rc" is a real constant. The parentheses and comma are part of the constant and must be present. The rules for the constituent real constants are given in Section 2.2.2.

A complex constant occupies four consecutive words of storage and is interpreted as a complex number.

Examples

```
(1.70391,-1.70391)
(+12739E3,0.)
```

2.2.5 Octal Constants

An octal constant is an alternate way of representing an integer constant and may be used in a like manner. When used in an arithmetic context, octal constants are treated as integer constants.

The general form for an octal constant is:

```
"nn
```

where nn is a string of octal digits.

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

An octal constant cannot be smaller than zero, nor greater than 177777 in FORTRAN IV and 37777777777 in FORTRAN IV-PLUS.

Examples

Valid Octal Constants	Invalid Octal Constants	
"7213	32767	(Double quote missing)
"1	"184	(Illegal character)
"17776		

2.2.6 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.

2.2.7 Hollerith Constants

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

Hollerith constants have the following general form:

$nHc_1c_2c_3 \ldots c_n$

where n is an unsigned, non-zero integer constant stating the number of characters in the string (including spaces), and each c_i is a printable ASCII character. The maximum number of characters is 255.

Hollerith constants are stored as byte strings, one character per byte.

Examples

Valid Hollerith Constants	Invalid Hollerith Constant
16HTODAY'S DATE IS: 1HΔ	3HABCD (Wrong number of characters)

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

The general 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 255.

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

Examples

```
'CHANGE PRINTER PAPER TO PREPRINTED FORM NO. 721'

'TODAY''S DATE IS:  '
```

2.2.8 Radix-50 Constants

Radix-50 is a special character data representation in which up to three characters from the Radix-50 character set (a subset of the ASCII character set) can be encoded and packed into a single PDP-11 storage word. Radix-50 constants may be used in DATA statements to initialize real variables and array elements.

A Radix-50 constant has the following form:

$nRc_1c_2...c_n$

where n is an unsigned non-zero integer constant that states the number of characters to follow, and each c_i is a character from the Radix-50 character set. The maximum number of characters is six. The character count must include any spaces that appear in the character string (the space character is a valid Radix-50 character).

The internal numeric value of any combination of one, two, or three Radix-50 characters may be found in Appendix A.

The Radix-50 characters and their code values are as follows:

Character	Radix-50 Value (Octal)
Space	0
A - Z	1-32
$	33
.	34
(not used)	35
0 - 9	36-47

Examples

3RABC

6RΔΔTOΔΔ

4RDK0: (Invalid; the colon is not a Radix-50 character)

2.3 VARIABLES

A variable is a symbolic name that is associated with a storage location. The value of the variable is the value currently stored in that location; that value can be changed by altering the contents of the storage location. (The form of a symbolic name is given in section 2.0.)

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 assigning 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.

2.3.1 Data Type by Definition

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

```
COMPLEX VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that the variable VAR1 is to be associated with a 4-word storage location that is to contain complex data, and that the variable VAR2 is to be associated with a 4-word double precision storage location.

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 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.

2.3.2 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

2.3.3 Assigning Hollerith Data to Variables

Hollerith data has no data type. However, Hollerith data can be assigned to variables of any type by DATA or input statements. Once assigned, the Hollerith data is treated as though it were data of the type established for the variable.

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 2-1. Each character occupies one byte of storage.

Hollerith data is stored as byte strings. 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. An attempt to assign more than the maximum number of characters causes the excess characters to be lost.

2.3.4 LOGICAL*1 Variables

A variable defined as LOGICAL*1 type represents a 1-byte storage area and can therefore contain only the numbers -128 through +127, or a single Hollerith character. It can also represent a logical value .TRUE. or .FALSE..

LOGICAL*1 logical and masking operations are performed on one byte of data at a time.

2.4 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 seven 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 the page contained several columns of figures, that page would 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 7.2),
2. DIMENSION statement (section 7.3), and
3. COMMON statement (section 7.4).

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.)

2.4.1 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:

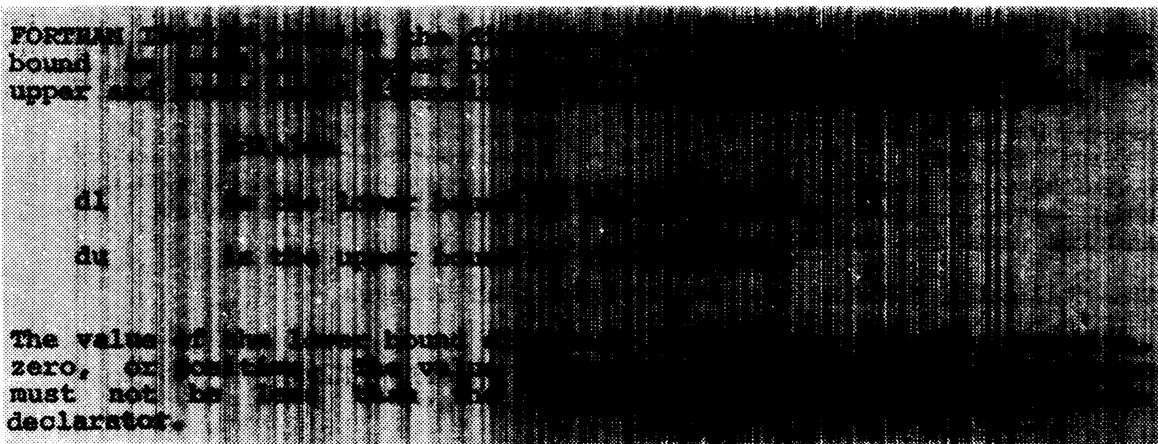
```
a (d[,d] ...)
```

a — is the symbolic name of the array -- the array name. (The form of a symbolic name is given in Section 2.0.)

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 7.

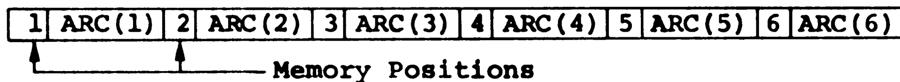
The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50 indicates that the dimension contains 50 elements. The dimension declarators may be constant or variable.



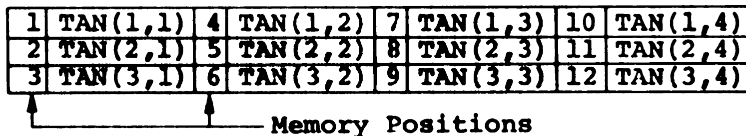
2.4.2 Array Storage

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)



2-Dimensional Array TAN(3,4)



3-Dimensional Array COS(3,3,3)

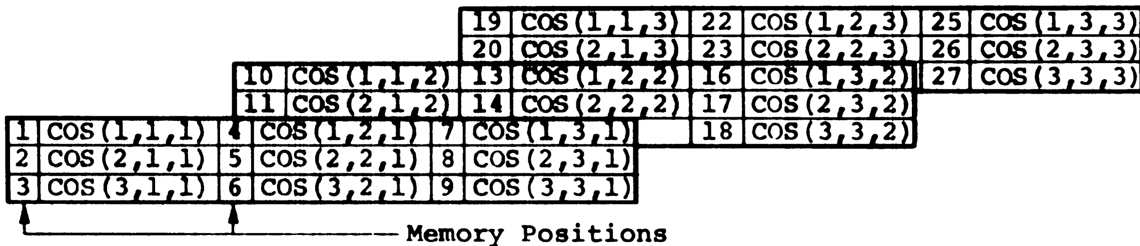


Figure 2-1
Array Storage

2.4.3 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 4-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.

The compiler stores LOGICAL*1 array elements in adjacent bytes.

2.4.4 Subscripts

A subscript qualifies an array name. A subscript is a list of subscript expressions enclosed in parentheses that determines which element in the array is referenced. The subscript is appended to the array name it qualifies.

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 COS in Figure 2-1 (which is the element occupying memory position 16).

```
COS(1,3,2)
```

Each subscript expression may be any valid arithmetic expression. If the value of a subscript expression is not an integer, it is converted to integer type before use.

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

Type declaration statements

COMMON statement

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 the entire array is to be used (or defined). The use of unsubscripted array names (except as arguments to subprograms) 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 7.5.

2.5 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.

2.5.1 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.)

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 + and - symbols may also be used as "unary operators" when written immediately preceding an arithmetic element to denote 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 must be assigned to a variable before its name can be used in an arithmetic expression.

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

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

In addition, a negative element can only be exponentiated by an integer element; 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 in this case is double precision.

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
* and /	Second
+ and - and Unary Operators	Third

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).

2.5.1.1 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
  ↑   ↑   ↑   ↑
  2   1   4   3

(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.

2.5.1.2 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
Logical	1 (Low)
Integer	2
Real	3
Double Precision	4
Complex	5 (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. Real operations - Real operations are performed only on real elements or a combination of real and integer elements. Any 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.

3. Double Precision operations - Any real 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 0.3333333000000000, not 0.3333333333333333. 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.

4. Complex operations - In an operation that contains any complex element, integer elements are converted to real type as previously described. Double precision elements are converted to real type by the rounding of the least significant portion. The real elements thus obtained are each designated as the real part of a complex number; the

imaginary part is zero. The expression is then evaluated using complex arithmetic and the resulting value is of type complex.

2.5.2 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.

Complex expressions can be related by the .EQ. and .NE. operators only. (If one complex expression is present, the other is converted to complex type.) Complex entities are equal if their corresponding real and imaginary parts are both equal.

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.

2.5.3 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. 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.
.EQV.	A .EQV. B	Logical equivalence. The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.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 logical value is first converted to an integer value, then the operation is carried out as for two integer elements. The resulting type is integer.

Evaluation of a logical expression is performed according to an order of precedence assigned to its operators. Some logical expressions can be evaluated without evaluating all sub-expressions; for example, if A is .FALSE. then the expression A .AND. (F(X,Y) .GT. 2.0) .AND. B is .FALSE.. The value of the expression can be determined by testing A without evaluating F(X,Y). With this method of evaluation, the function subprogram F is not necessarily called and side-effects resulting from the call may not occur.

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

Operator	Evaluated
**	First
*,/	Second
+,- and Unary Operators	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.	Eighth

The delimiting decimal points of logical 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 consecutively, 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.

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 on the right of the equal sign to the variable or array element on the left of the equal sign. The previous value of the variable, if any, is lost.

The arithmetic assignment statement has the following form:

```
v = e
```

v   is a numeric variable name or array element name.

e   is an expression.

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".

ASSIGNMENT STATEMENTS

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

The expression must yield a value that conforms to the requirements of the variable or array element to which it is to be assigned (for example, a real expression that produces a value greater than 32767 is unacceptable if the entity on the left of the equal sign is an INTEGER*2 variable).

If the data type of the variable or array element on the left of the 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.

Table 3-1
Conversion Rules for Assignment Statements

VARIABLE OR ARRAY ELEMENT (V)	EXPRESSION (E): INTEGER, LOGICAL, HOLLERITH, OR OCTAL CONSTANT	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	Assign E to V	Truncate E to Integer and assign to V	Truncate E to Integer and assign to V	Truncate real part of E to integer and assign to V; imaginary part of E is not used
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	Assign real part of E to V; imaginary part of E is not used
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	Assign real part of E to MS portion of V; LS portion of V is zero, imaginary part of E is not used
COMPLEX	Append fraction (.0) to E and assign to real part of V; imaginary part of V is 0.0	Assign E to real part of V; imaginary part of V is 0.0	Assign MS portion of E to real part of V; imaginary part of V is 0.0, LS portion of E is rounded	Assign E to V

MS = Most Significant (high-order)

LS = Least Significant (low-order)

Examples

Valid Statements

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

PI = 3.14159

SUM = SUM+1.
```

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.)

3.2 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 equal sign and assigns the resulting logical value to the variable or array element on the left.

The form of the logical assignment statement is shown below:

v = e

v is a variable or array element of type Logical.

e is a logical expression.

The variable or array element on the left of the equal sign must have been previously defined as being of logical type by a data type declaration statement or an IMPLICIT statement. Its value may be initially undefined.

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

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).

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. (It must not be the label of a FORMAT 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 statement(s) in which the assigned variable is to be used. The ASSIGN statement and the assigned GO TO statement(s) must occur in the same program unit.

Consider the following example. In this example, the statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with statement 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 which 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 expression within the statement.

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

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

slist is a list of one or more executable statement labels separated by commas. The list of labels is called the transfer list.

e is an arithmetic expression 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 is optional; the commas within the list, and the parentheses, are required.

The computed GO TO statement evaluates the expression e and, if necessary, converts the resulting value to integer type. The GO TO statement then transfers control to the e'th statement label in the transfer list. That is, if the list contains (30,20,30,40), and the value of e is 2, the GO TO statement passes control to statement 20, and so on.

If the value of the expression is less than 1, or greater than the number of labels in the transfer list, 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) SITU(J,K)+1
```

In the second example, if the value of the expression is 1 (the value of array element SITU(J,K) is zero), the GO TO statement transfers control to statement 320; if the value of the expression 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 GO

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.

In [illegible]
in the [illegible]
statement [illegible]

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) s1, s2, s3
```

e — is an arithmetic expression.

s — is a statement label 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.

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 s1
Equal to 0	Label s2
Greater than 0	Label s3

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.

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 logical 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, 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
```

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=e1,e2[,e3]
```

```
DO s[,]i=e1,e2[,e3]
```

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

i is an integer variable.

e1,e2,e3 are integer expressions.

The variable i is called the control variable of the DO and e1,e2,e3 are called the initial, terminal, and increment parameters respectively. If the increment parameter is omitted, a default increment value of 1 is used.

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.

In FORTRAN IV-PLUS, the control [illegible] INTEGER*2, INTEGER*4, REAL, or DOUBLE [illegible] and increment parameters may be of [illegible] before use, to the type of the control variable, if necessary.

The DO statement first evaluates the expressions e1, e2, e3 to determine values for the initial, terminal, and increment parameters.

The value of the initial parameter is then assigned to the control variable. The executable statements in the range of the DO loop are then executed repeatedly.

The number of executions of the DO range (the iteration count) is given by

$$\left[\frac{e2 - e1}{e3} \right] +1$$

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 or negative, 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. 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, 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.)

In [illegible]
modified [illegible]

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 Sections 4.3.2 and 4.3.3.

Examples

DO 100 K=1,50,2	(25 iterations, K=49 during final iteration)
DO 350 J=50,-2,-2	(27 iterations,J=-2 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 the same terminal statement.

<table>
<tr><th>Correctly Nested
DO Loops</th><th>Incorrectly Nested
DO Loops</th></tr>
<tr><td><pre>
      DO 45 K=1,10
      .
      .
      DO 35 L=2,50,2
      .
      .
   35 CONTINUE
      .
      .
      DO 45 M=1,20
      .
      .
   45 CONTINUE
</pre></td><td><pre>
      DO 15 K=1,10
      .
      .
      DO 25 L=1,20
      .
      .
   15 CONTINUE
      .
      .
      DO 30 M=1,15
      .
      .
      .
   25 CONTINUE
      .
      .
   30 CONTINUE
</pre></td></tr>
</table>

Figure 4-1
Nesting of DO Loops

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.

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
Extended  .
Range     .
        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

In FORTRAN IV-PLUS, the following rules govern the use of a DO statement extended range:

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

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

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

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

5. 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 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, alphanumeric literals, or subprogram 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.

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.

Examples

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

CALL PNTOUT

CALL EXIT
```

(EXIT is a system subroutine that terminates the execution of the program and returns control to the operating system.)

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
```

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 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
```

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 a decimal digit string containing one to five digits, an alphanumeric literal, or an octal 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.

Examples

```
PAUSE "13731

PAUSE 'MOUNT TAPE REEL #3'
```

4.8 STOP STATEMENT

The STOP statement is used to terminate program execution.

The STOP statement appears in the following form:

```
STOP [disp]
```

disp is a decimal digit string containing one to five digits, an alphanumeric literal, or an octal 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.

Examples

```
      STOP 98

      STOP "7777

99999 STOP

      STOP 'END OF RUN'
```

4.9 END STATEMENT

The END statement marks the 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

INPUT/OUTPUT STATEMENTS

5.1 OVERVIEW

Input of data by a FORTRAN program is performed by READ and ACCEPT 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. The TYPE and ACCEPT statements have no such reference, as they cause data transfer between the processor and an implicit logical unit that is normally associated with the user's terminal. Similarly, the PRINT statement outputs data to an implicit logical unit that is normally associated with the line printer.

READ and WRITE statements fall into the following categories:

1. Unformatted Sequential I/O

   Unformatted sequential READ and WRITE statements transmit binary data without translation.

2. Formatted Sequential I/O

   Formatted sequential READ and WRITE statements contain references to FORMAT statements, or to format specifications in arrays, 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 DEFINE FILE statement or an OPEN statement.

FORTRAN IV-PLUS provides a fourth [illegible] statements:

4. **Formatted Direct Access I/O**

   Formatted direct access [illegible] references to FORMAT [illegible] in arrays, and perform [illegible] direct access files.

Any type of READ or WRITE 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.

FORTRAN IV-PLUS provides two [illegible] OPEN and CLOSE.

1. **The OPEN statement [illegible] logical unit and a file or device; it [illegible] needed for READ and WRITE operations.**

2. **The CLOSE statement [illegible] file or device and a logical unit.**

5.1.1 Input/Output Devices

PDP-11 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 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 associated FORTRAN User's Guide for additional information.

5.1.2 Format Specifiers

Format specifiers are 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. Chapter 6 discusses FORMAT statements in detail.

5.1.3 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. 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.

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 additional records can be transmitted.

5.2 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 a unit. In addition, the I/O list of an output statement may contain constants and expressions. 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.

5.2.1 Simple Lists

A simple I/O list consists of a single variable, array reference, or expression, 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 or ACCEPT 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,X,I1,X,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.

Any valid expression can be included in an output statement I/O list. However, the expression must not cause further I/O operations to be attempted. A reference in an output statement I/O list expression to a function subprogram that itself performs some form of I/O is an example of this prohibited case.

An expression must not be included in an input statement I/O list except as a subscript expression in an array reference.

5.2.2 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 of 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. 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 the reverse of the 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 one through twenty.

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). An expression may be used for the initial, terminal, or increment parameter of an implied DO list, as long as it conforms to the rules in section 4.3.

5.3 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.

5.3.1 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][,ERR=s])[list]
```

u is a logical unit number.

list is an I/O list.

s 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.)

5.3.2 Unformatted Sequential WRITE Statement

The unformatted sequential WRITE statement has the following form:

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

u is a logical unit number.

s is an executable statement label.

list is an I/O list.

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.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified device.

Examples

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

```
        WRITE (4)                (Writes a null record on logical
                                 unit 4.)
```

5.4 FORMATTED SEQUENTIAL INPUT/OUTPUT

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

5.4.1 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 by the READ statement translate the data from external to internal format.

The formatted sequential READ statement appears in one of the following forms:

```
READ  f[,list]

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

s   is an executable statement label.

u   is a logical unit number.

f   is a format specifier.

list is an I/O list.

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)
```

A statement of the form:

```
      READ 200, ALPHA,BETA,GAMMA
```

causes data to be read from a system dependent logical unit.

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

5.4.2 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 specifications in the associated FORMAT statement or format array.

The formatted sequential WRITE statement appears as:

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

u is a logical unit number.

f is a format specifier.

s is an executable statement label.

list is an I/O list.

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.

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.

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

Statement	Description
`WRITE (6, 650)` `650 FORMAT (' HELLO, THERE')`	(Outputs contents of FORMAT statement to logical unit 6.)
`WRITE (1,95) AYE,BEE,CEE` `95 FORMAT (F8.5,F8.5,F8.5)`	(Writes one record of three fields to logical unit 1.)
`WRITE (1,950) AYE,BEE,CEE` `950 FORMAT (F8.5)`	(Writes three separate records of one field each to logical unit 1.)

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.

5.4.3 ACCEPT Statement

The function of the ACCEPT statement is identical to that of the formatted READ statement, except that input is read from a logical unit normally associated with the terminal keyboard.

The form of the ACCEPT statement is:

```
ACCEPT f[,list]
```

f is a format specifier.

list is an I/O list.

The rules for the format reference and I/O list of an ACCEPT statement are the same as those for the formatted READ statement (Section 5.4.1).

Examples

```
    ACCEPT 100, NUMBER            (Accepts one Integer value from
100 FORMAT (I4)                   terminal keyboard.)

    ACCEPT 200                    (Reads 13 characters from
200 FORMAT ('PUT DATA HERE')      keyboard, places them in FORMAT
                                  statement.)
```

5.4.4 TYPE Statement

The TYPE statement functions identically to the formatted WRITE statement except that output is directed to a logical unit normally connected to the terminal printer.

The TYPE statement has the following form:

```
TYPE f[,list]
```

f is a format specifier.

list is an I/O list.

The rules for the format reference and I/O list of a TYPE statement are the same as those for the formatted WRITE statement (Section 5.4.2).

Examples

```
    TYPE FACE, BOLD                     (Displays contents of BOLD
                                        on terminal in the format
                                        specified by contents of
                                        array FACE.)

    TYPE 400                            (Types message from FORMAT
400 FORMAT (' MOUNT NEW TAPE REEL')     statement on terminal.)
```

5.4.5 PRINT Statement

The function of the PRINT statement is the same as that of the formatted WRITE statement and TYPE statement, except that output is directed to a logical unit normally associated with a line printer.

The PRINT statement takes the form:

```
PRINT f[,list]
```

f is a format specifier.

list is an I/O list.

The format reference and I/O list in a PRINT statement follow the same rules as specified for the formatted sequential WRITE statement (section 5.4.2).

Examples

```
     PRINT 999, NPAGE                (Prints page number in upper
999  FORMAT (1H1,100X,'PAGE  ',I3)   right-hand corner of new
                                     page.)

     PRINT 222                       (Prints contents of FORMAT
222  FORMAT (' END OF LISTING')      statement on line printer.)
```

5.5 UNFORMATTED DIRECT ACCESS INPUT/OUTPUT

Unformatted direct access READ and WRITE statements are used to perform direct access I/O on any directory-structured device. The DEFINE FILE 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.

5.5.1 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]
```

u   is a logical unit number.

r   is an integer expression that specifies the record number.

s   is an executable statement label.

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.

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 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 previously 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, 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, assigns five Real values to array RHO.)

5.5.2 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]
```

u is a logical unit number.

r is an integer expression that specifies the record number.

s is an executable statement label.

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.

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 entire contents of ARRAY to a file on logical unit 3 into the record indicated by value of J.)

5.5.3 DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of a file upon which direct access I/O is to be performed.

The DEFINE FILE statement appears as:

```
DEFINE FILE u (m,n,U,v) [,u(m,n,U,v)]...
```

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

m is an integer constant or integer variable that specifies the number of records in the file.

n is an integer constant or integer variable that specifies the length, in words, of each record.

U specifies that the file is unformatted (binary). The letter U is the only acceptable entry in this position.

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.

The DEFINE FILE statement specifies that a file containing m fixed-length records of n words each exists, or is to exist, on logical unit u. The records in the file are sequentially numbered from 1 through m.

The DEFINE FILE statement must be executed before the first direct access I/O statement that refers to the specified file.

The DEFINE FILE statement also 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 or a FIND 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.

Example

```
DEFINE FILE 3 (1000,48,U,NREC)
```

This specifies that logical unit 3 is to refer to a file of 1000 fixed-length records, each record of which is 48 words long. The records are numbered sequentially from 1 through 1000, and are in binary format. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the one just processed.

f is a format specifier.

s is an executable statement [illegible]

list is an I/O list.

If the list and format specification [illegible] than a record contains, all of the [illegible] and an error condition exists.

5.6.2 Formatted Direct Access [illegible]

The formatted direct access [illegible] record in the direct access [illegible] to the unit. If the list and format [illegible] records, an error condition [illegible] values which are converted to [illegible] by the format specification.

The statement has the following [illegible]

[illegible]

u [illegible]

r is an integer expression [illegible]

f is a format specifier.

s is an executable statement [illegible]

list is an I/O list.

If the [illegible]
blank [illegible]

If the [illegible]
fit into [illegible]

5.7 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 the end of the file. These specifications appear as follows:

```
END=s
```

and

```
ERR=s
```

s is the statement label of an executable statement to which control is to be transferred.

A READ or WRITE statement may contain either or both of the above specifications, in either order. 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 or WRITE statements. An END=S specification may be included in direct access READ or WRITE 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.

The FORTRAN User's Guide describes system subroutines that may be used to control processing of error conditions. These subroutines obtain information from the I/O system concerning the error and serve as an aid in determining what corrective action is to be taken.

5.8 AUXILIARY INPUT/OUTPUT STATEMENTS

The statements in this category are used to perform file management functions.

5.8.1 REWIND Statement

The REWIND statement causes a currently open file to be repositioned to the beginning of the file.

The form of the REWIND statement follows:

```
REWIND u
```

u   is a logical unit number.

The unit number in the REWIND statement must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example

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

5.8.2 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.

The unit number must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example

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

5.8.3 ENDFILE Statement

The ENDFILE statement writes an end-file record on a currently open sequential file.

The ENDFILE statement is written as follows:

```
ENDFILE u
```

u is a logical unit number.

A file must be open on unit u.

Example

```
ENDFILE 2
```

The above statement writes an end-file record to the currently open file on logical unit two.

5.8.4 FIND Statement

The FIND statement positions a direct access file on a specified unit to a particular record and sets the associated variable of the file to that record number. No data transfer takes place.

The form of the FIND statement is:

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

u is a logical unit number.

r is a record number.

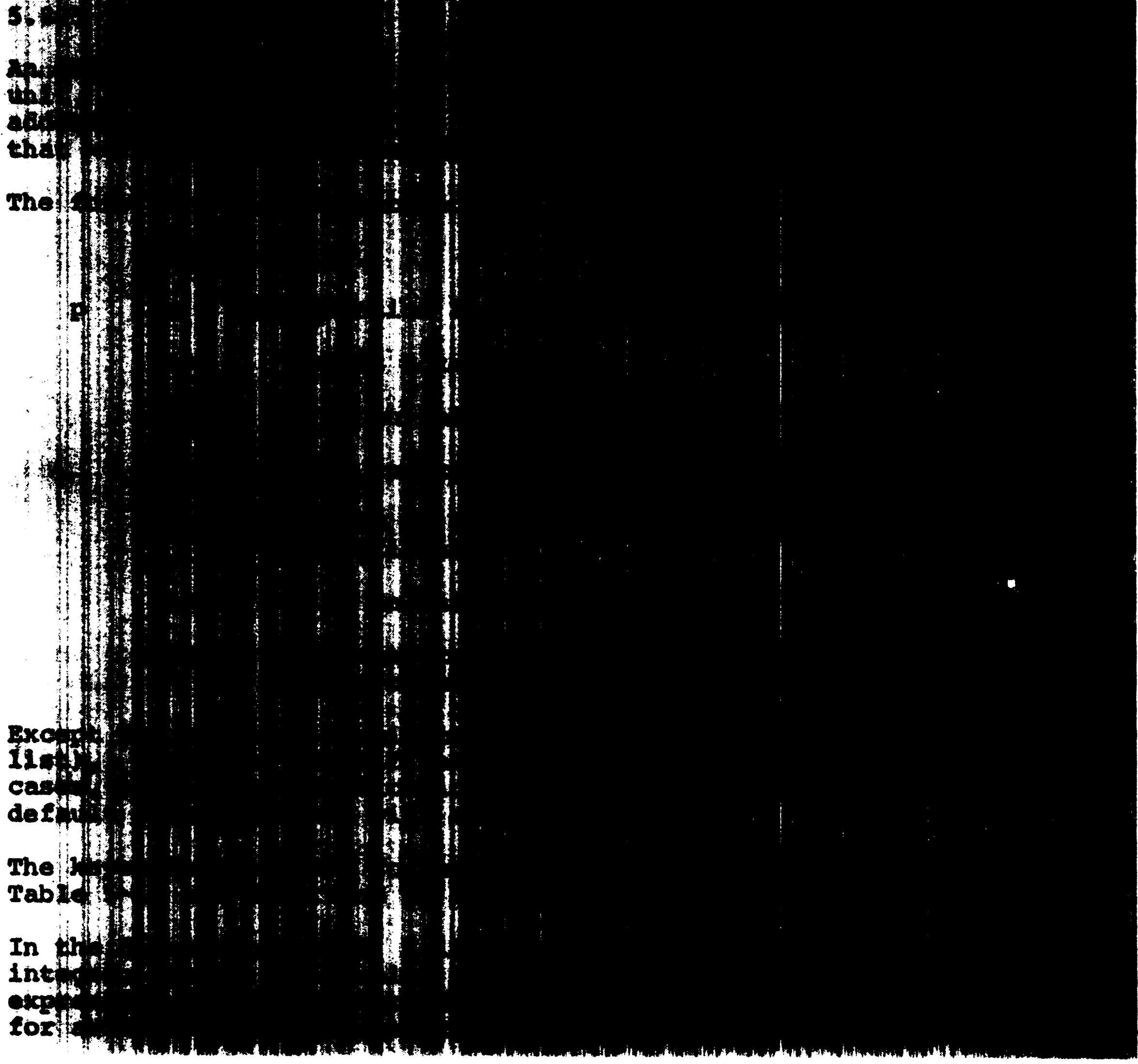
The unit number in the statement must refer to a directory-structured device. The file that is to be affected by this statement must have been previously defined for direct access processing.

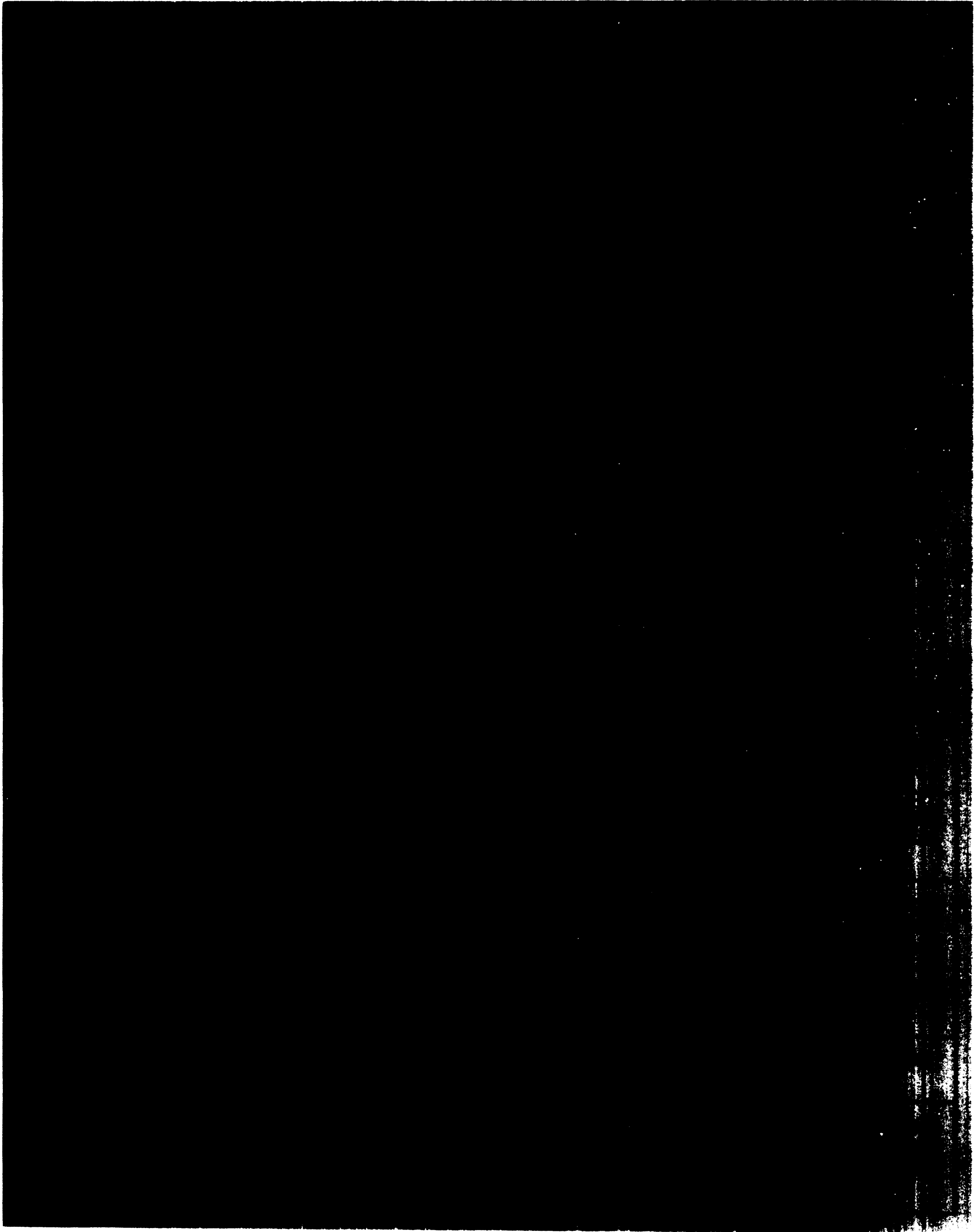
The record number must not be less than 1 nor greater than the number of records in the file.

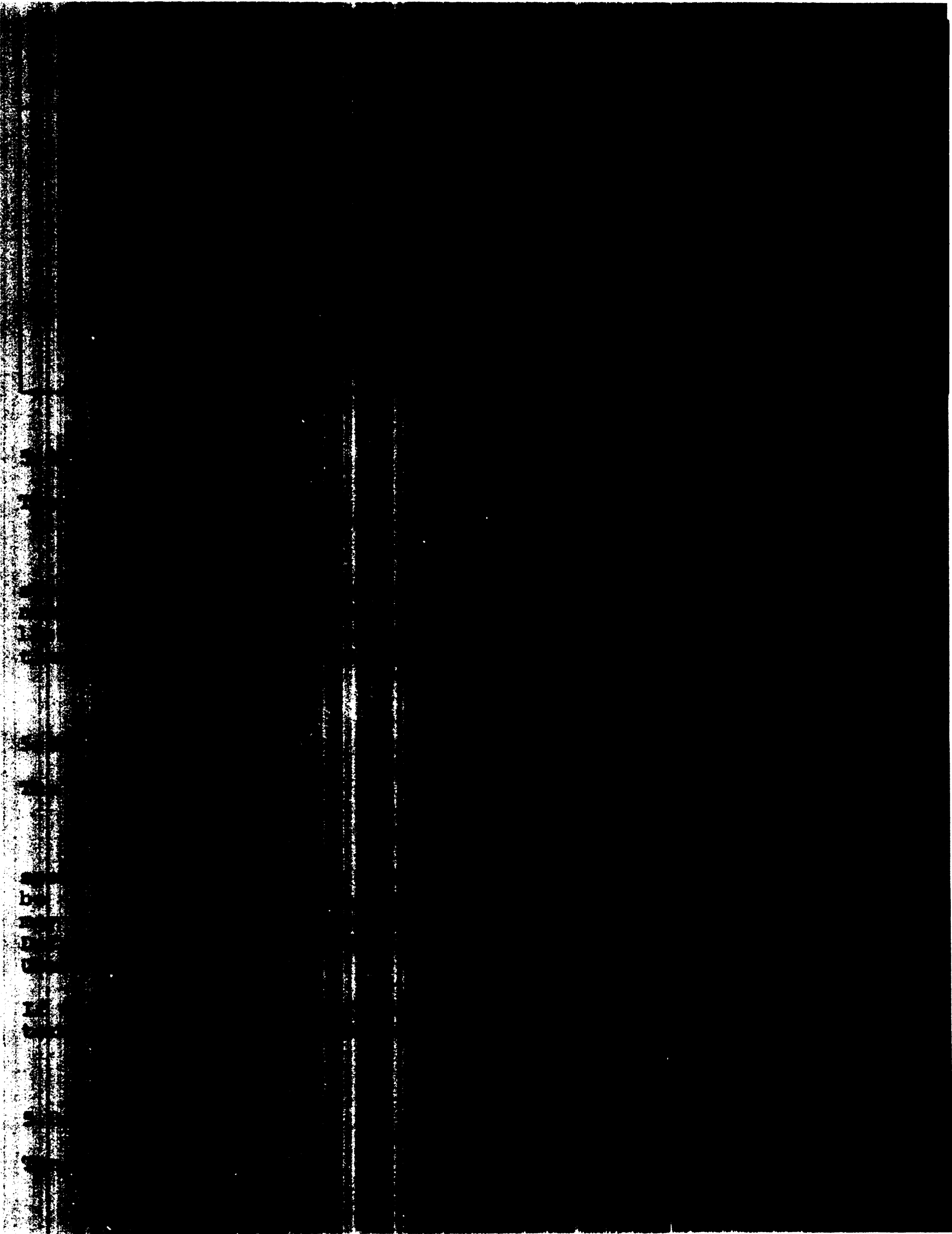
Examples

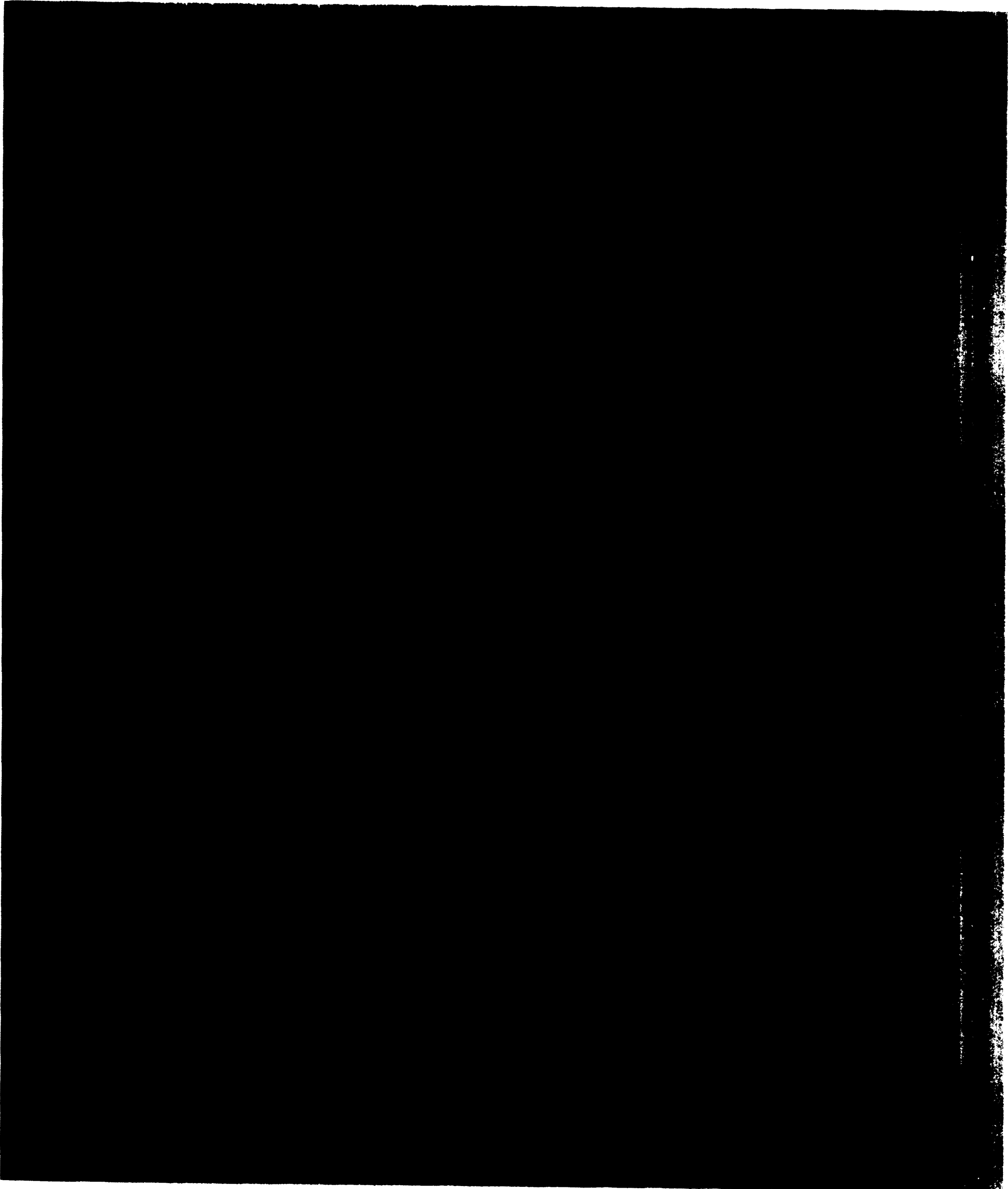
FIND (1'1)	Positions logical unit 1, and its associated variable, to the first record of the file.

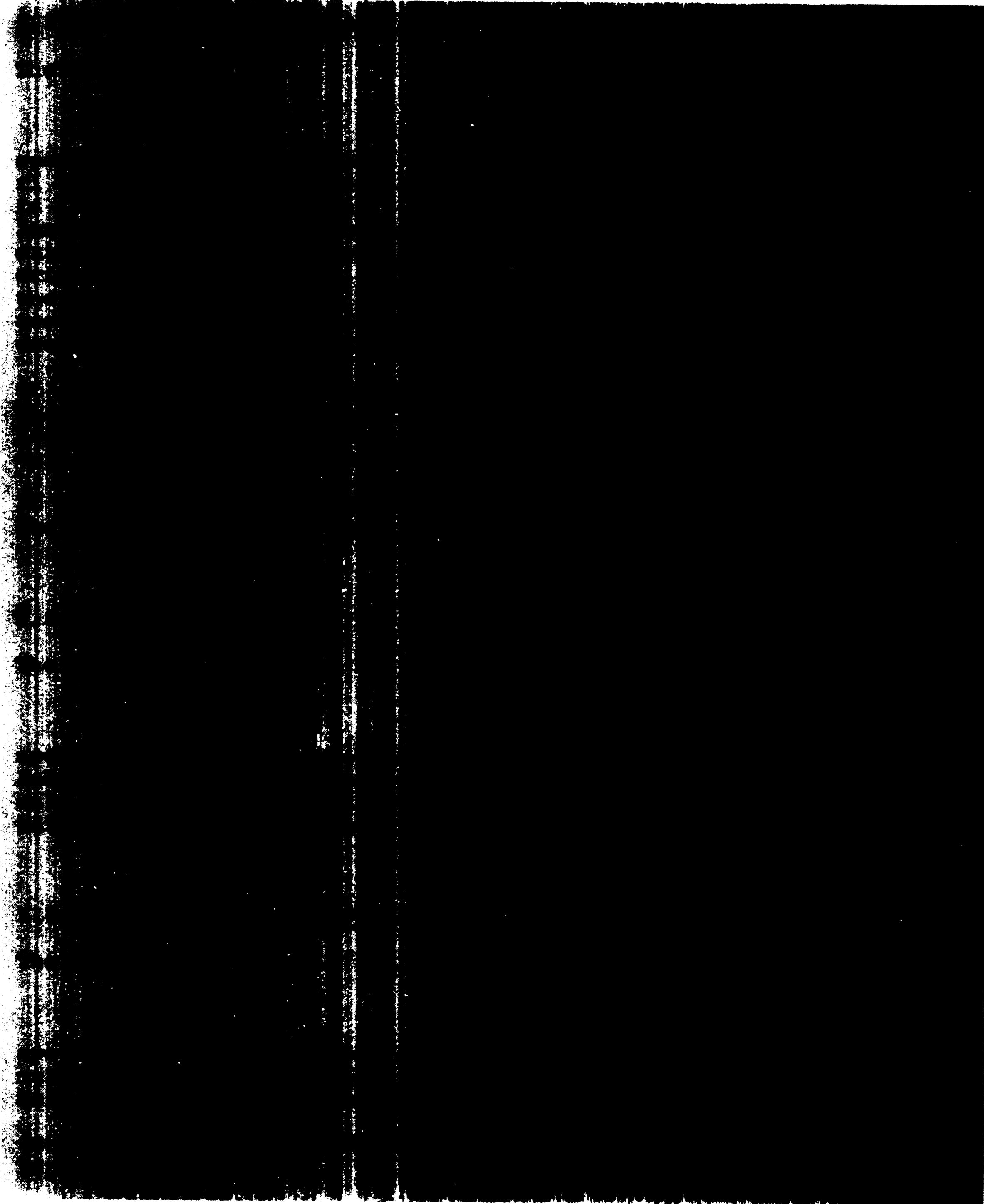
FIND (4'INDX)	Positions file and associated variable to record identified by contents of INDX.

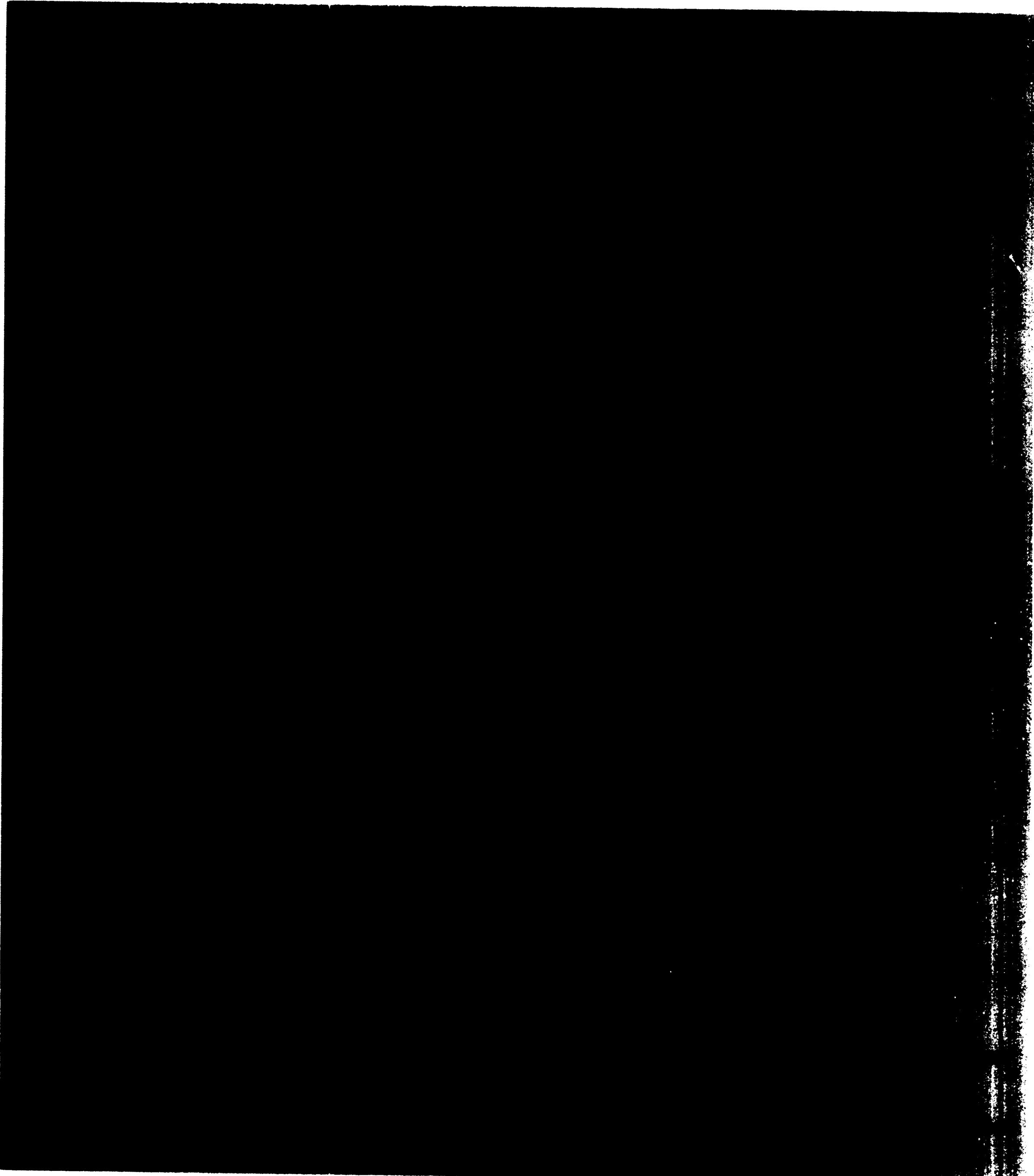


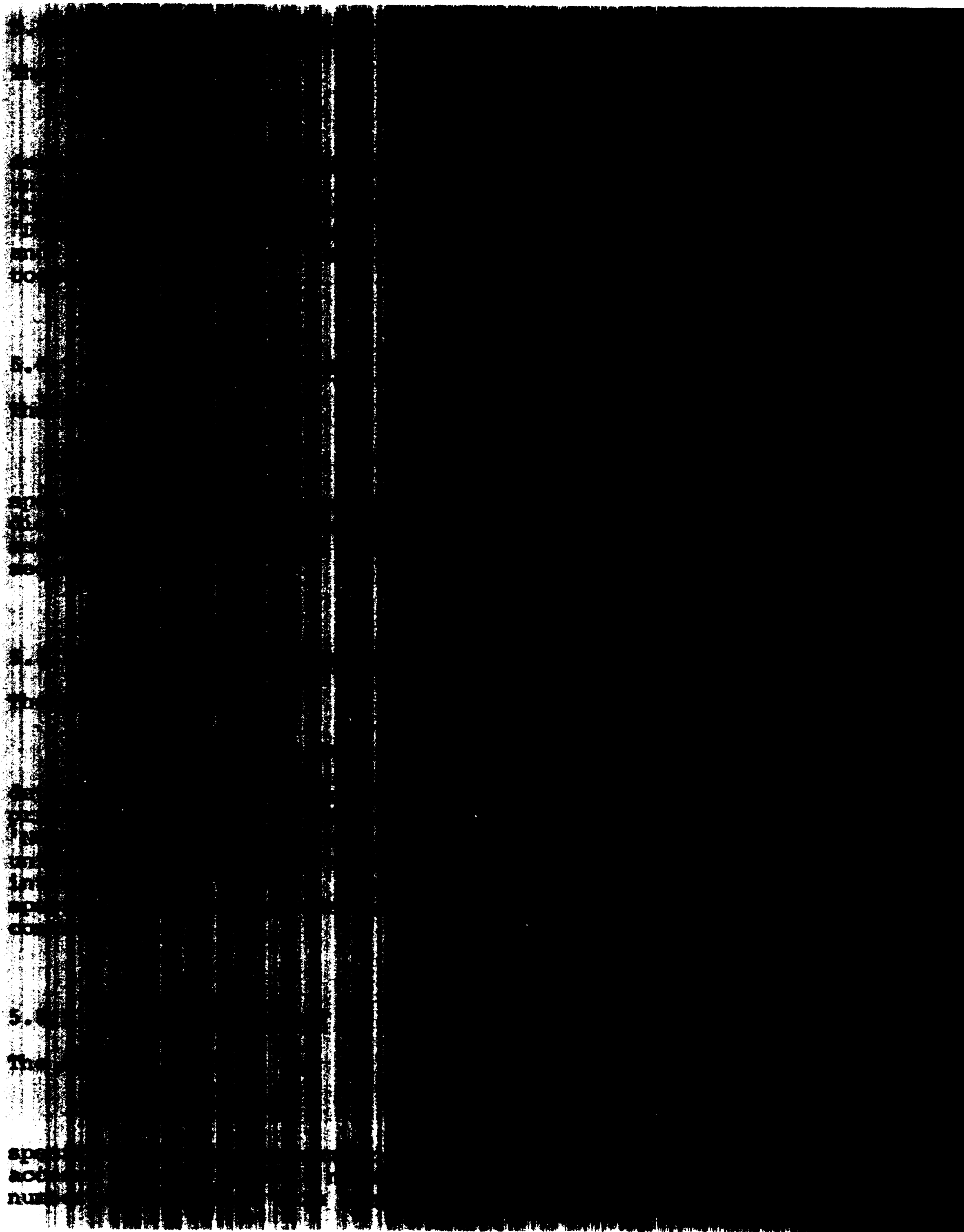


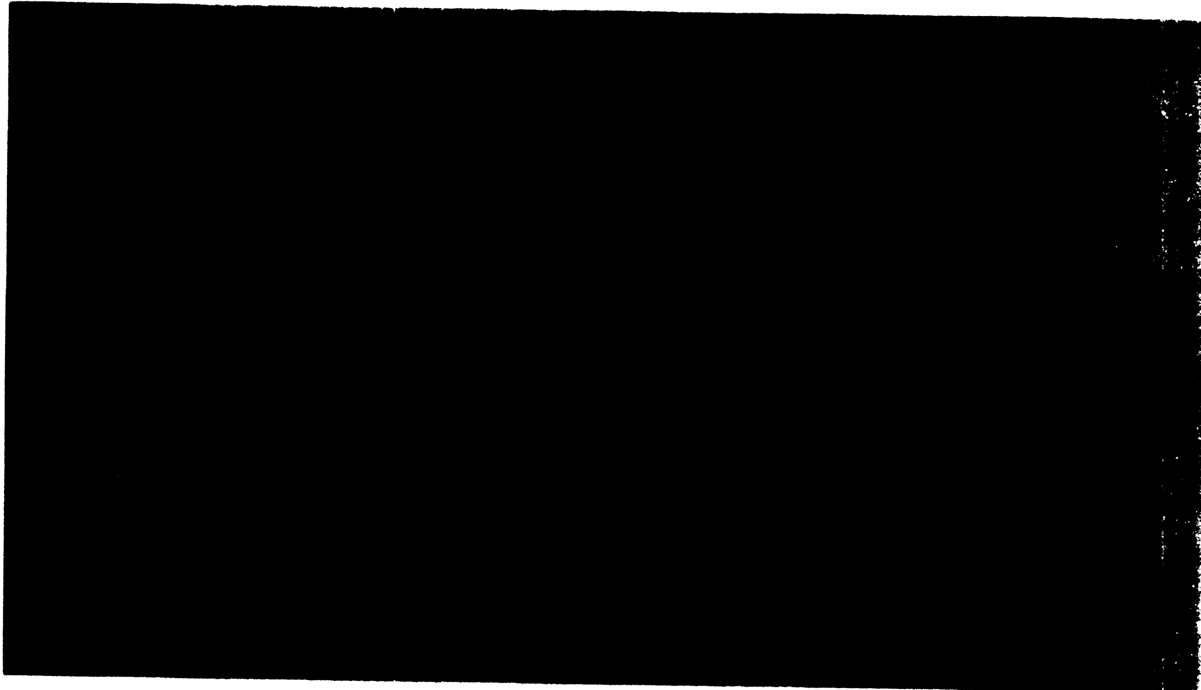












5.9 ENCODE AND DECODE STATEMENTS

These two statements perform data transfers according to format specifications, 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,f,b) [list]

DECODE(c,f,b) [list]
```

c is an integer expression representing the number of ASCII characters (bytes) that are to be converted or that are to result from the conversion. (This is analogous to the length of an external record.)

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.

b is the name of an 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.

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 or format array to ASCII format and stores those ASCII characters in array b. 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.

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 b, processing the array in the order of subscript progression, converts that data to the form specified by the FORMAT statement or format array, and assigns the data to the elements of the I/O list from left to right.

In both [illegible] and [illegible] [illegible]
the array [illegible] b to cause a [illegible]

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 6

FORMAT STATEMENTS

6.1 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.

The FORMAT statement is written:

FORMAT $(q_1 f_1 s_1 f_2 s_2 \cdots f_n q_n)$

where each f is a field descriptor, or a group of field descriptors enclosed in parentheses, each s is a field separator and each q is zero or more slash (/) field separators. The entire list of field descriptors and field separators including the parentheses is called the format specification. The list must be enclosed in parentheses.

A field descriptor in a format specification appears in one of the following forms:

rCw

or

rCw.d

where C is a format code; w 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. See section 6.2.16 for further discussion of field repetition.

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 6.4.

The field descriptors used in format specifications are as follows:

1. Integer: Iw, Ow

2. Logical: Lw

3. Real, Double Precision, Complex: Fw.d, Ew.d, Dw.d, Gw,d

4. Literal and editing: Aw, nH, '...', nX, Tn, Q

(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 in the range -127 to +127 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 6.2.15.

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 6.7.

6.2 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.

6.2.1 I Field Descriptor

The I field descriptor governs the translation of integer data. It appears as:

Iw

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; 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

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 any are 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

6.2.2 O Field Descriptor

The O field descriptor governs the transmission of octal values. It appears as follows:

Ow

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 or logical type. The external field must contain only the numerals 0 through 7; it must not contain a sign, a decimal point, or an exponent field. For example:

Format	External Field	Internal Octal Representation
O5	32767	32767
O4	16234	1623
O6	13ΔΔΔΔ	130000
O3	97Δ	not permitted; error

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 spaces are inserted; if the data exceeds the field width, the entire field is filled with asterisks. 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 or logical type. For example:

Format	Internal (Decimal) Value	External Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O5	13.52	not permitted; error

6.2.3 F Field Descriptor

The F field descriptor specifies the data conversion and editing of real or double precision values, or the real or imaginary parts of complex 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

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 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. For example:

Format	External Field	Internal Representation
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

On output, the F field descriptor causes the value of the related I/O list element to be rounded to d decimal positions and transmitted to an external field w characters in length, right justified. If the converted data consists of fewer than w characters, leading spaces are inserted; 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 any are expected (plus signs are suppressed), at least one digit to the left of the decimal point, 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	**
F10.4	-23.24352	ΔΔ-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

6.2.4 E Field Descriptor

The E field descriptor handles the transmission of real data in exponential format. It appears as follows:

Ew.d

The corresponding I/O list element must be of real, double precision, or complex type.

The E field descriptor causes a READ or ACCEPT 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. 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, TYPE, or PRINT 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:

EΔnn

or

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 any are 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	Δ0.48EΔ06
E12.5	475867.222	Δ0.47587EΔ06
E12.3	0.00069	ΔΔΔ0.690E-03
E10.3	-0.5555	-0.556EΔ00
E5.3	56.12	*****

6.2.5 D Field Descriptor

The D field descriptor governs the transmission of real or double precision data. It appears as follows:

Dw.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

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	ΔΔΔΔΔ0.541387625793DΔ04
D9.6	1.2	*********

6.2.6 G Field Descriptor

The G field descriptor transmits real, double precision, or complex data in a form that is in effect a combination of the F and E field descriptors. It appears as follows:

Gw.d

On input, the G field descriptor functions identically to the F field descriptor (see Section 6.2.3).

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 6-1.

Table 6-1
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m < 0.1	Ew.d
0.1 ≤ m < 1.0	F(w-4).d, 4X
1.0 ≤ m < 10.0	F(w-4).(d-1), 4X
. . .	. . .
10d-2 ≤ m < 10d-1	F(w-4).1, 4X
10d-1 ≤ m < 10d	F(w-4).0, 4X
m ≥ 10d	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 6.2.10.

The field width, w, must include space for a minus sign, if any are 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	-1234.56789012	Δ-1234.57ΔΔΔΔ
G13.6	12345.67890123	ΔΔ12345.7ΔΔΔΔ
G13.6	123456.78901234	ΔΔ123457.ΔΔΔΔ
G13.6	-1234567.89012345	-0.123457EΔ07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Values	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	-1234.56789012	Δ-1234.567890
F13.6	12345.67890123	Δ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	*************

6.2.7 L Field Descriptor

The L field descriptor governs the transmission of logical data. It appears as:

Lw

The corresponding I/O list element must be of logical 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 if the entire field is blank, the value .FALSE. is assigned. Any other value in the external field causes an error condition.

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. 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

6.2.8 A Field Descriptor

The A field descriptor controls the transmission of alphanumeric data. It is written as shown below. (The corresponding I/O list element may be of any data type.)

Aw

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*1	1
Logical*2	2 (FORTRAN IV-PLUS only)
Logical*4	4
Integer*2	2
Integer*4	4
Real	4
Double Precision	8
Complex	8

If w is greater than the maximum number of characters that can be stored in the corresponding I/O list element, only the rightmost one, two, four, or eight 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	
A6	PAGEΔ#	#	(Logical*1)
A6	PAGEΔ#	Δ#	(Integer*2)
A6	PAGEΔ#	GEΔ#	(Real)
A6	PAGEΔ#	PAGEΔ#ΔΔ	(Double Precision, Complex)

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	OHMS	ΔOHMS
A5	VOLTSΔΔΔ	VOLTS
A5	AMPERESΔ	AMPER

6.2.9 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:

```
      TYPE 100
100   FORMAT (41HΔENTERΔPROGRAMΔTITLE,ΔUPΔTOΔ20ΔCHARACTERS)
      ACCEPT 200
200   FORMAT (20HΔΔTITLEΔGOESΔHEREΔΔΔ)
```

The TYPE statement transmits the characters from the H field descriptor in statement 100 to the user's terminal. The ACCEPT 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; if the user enters fewer than 20 characters, the remainder of the H field descriptor is filled with spaces to the right.

6.2.9.1 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.

6.2.10 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 6.3.

6.2.11 T Field Descriptor

The T field descriptor, which appears as follows:

```
Tn
```

is a tabulation specifier. The value of n must be greater than or equal to one, but not greater than the number of characters allowed in the external record.

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 devices other than the line printer or terminal, 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 6.3) which is never printed. For example the statements:

```
      PRINT 25
25    FORMAT (T51,'COLUMNΔ2',T21,'COLUMNΔ1')
```

would cause the following line to be printed:

```
       Position 20                    Position 50
            ↓                              ↓
            COLUMN 1                       COLUMN 2
```

6.2.12 Q Field Descriptor

The Q field descriptor, which is simply the letter Q, is used to obtain the number of characters in the input record remaining to be transmitted during a READ operation. It is ignored on output. The I/O list element associated with the Q field descriptor must be of integer type.

As an example, the statements:

```
      READ (4,1000) XRAY,YANKEE,ZULU,NCHARS
1000  FORMAT (3F10.5,Q)
```

read three fields of ten characters each, assigning them as Real values to the variables XRAY, YANKEE, and ZULU. However, any or all of those fields may be terminated before ten characters have been read if one or more external field separators appears in the input record. The Q field descriptor causes the number of characters remaining in the input record to be assigned to the variable NCHARS. By placing the Q descriptor first in the format specification, the actual length of the input record may be determined.

When the Q field descriptor is used with a WRITE statement it has no effect except that the corresponding list item is skipped.

6.2.13 $ Descriptor

The character $ (dollar sign) appearing in a format specification modifies the carriage control specified by the first character of the record. The $ descriptor is intended primarily for interactive I/O and causes the terminal print position to be left at the end of the text written (rather than returned to the left margin) so that a typed response will appear on the same line following the output.

6.2.14 Complex I/O

Since a complex value is an ordered pair of real values, input or output of a complex entity is governed by two real field descriptors, using any combination of the forms Fw.d, Ew.d, Dw.d or Gw.d.

On input, two successive fields are read and assigned to a complex I/O list element as its real and imaginary parts, respectively. For example

Format	External Fields	Internal Representation
F8.5,F8.5	1234567812345.67	123.45678,12345.67
E9.1,F9.3	734.432E8123456789	734.432E8,12345.678

On output, the constituent parts of a complex value are transmitted under the control of repeated or successive field descriptors. Nothing intervenes between those parts unless explicitly stated by the format specification. For example:

Format	Internal Values	External Representation
2F8.5	2.3547188,3.456732	Δ2.35472Δ3.45673
E9.2,' , ',E5.3	47587.222,56.123	Δ0.48EΔ06Δ,Δ*****

6.2.15 Scale Factor

The location of the decimal point in real and double precision values, and in the constituent parts of complex 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 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	-270.139EΔ00
1PE12.3	-270.139	ΔΔ-2.701EΔ02
1PE12.2	-270.139	ΔΔΔ-2.70EΔ02
-1PE12.2	-270.139	ΔΔΔ-0.03EΔ04

6.2.16 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:

```
50    FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
                    \_________/ \_________/ \_________/
                         1           2           3
```

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.

6.2.17 Variable Format Expressions

An expression can be used in a [illegible] can be used (except as a [illegible] brackets. For example:

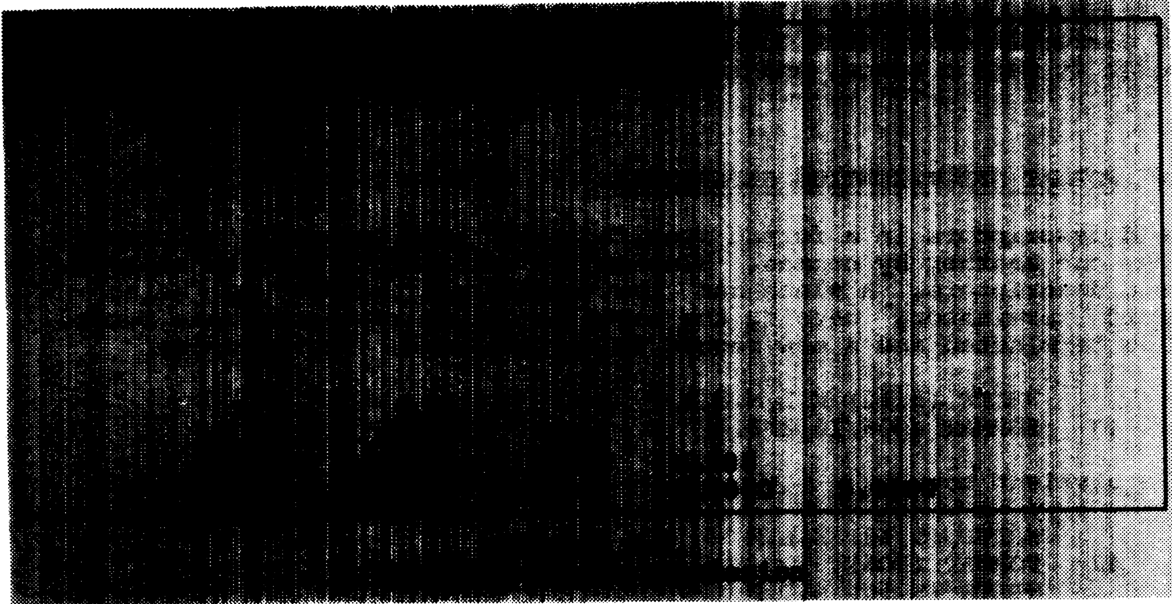
```
FORMAT (I<J+1>)
```

will cause an I conversion with a field [illegible] value of J at the time the format is [illegible] re-evaluated each time it is [illegible] If the expression is not of type [illegible] integer prior to use. Any valid [illegible] including function calls and [illegible]

A complete example is shown in Figure [illegible]

The value of a variable format [illegible] on magnitude applying to its use [illegible] will occur.

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./
C
      DO 10 I = 1,10
      WRITE (5,100) I
100   FORMAT(I<MAX0(I,5)>)
10    CONTINUE
C
      DO 20 I = 1,5
      WRITE (5,101) [illegible]
101   FORMAT (<I>F10.[illegible])
20    CONTINUE
      END
```



6.3 CARRIAGE CONTROL

The first character of every record transmitted to a printing device 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 6-2.

Table 6-2
Carriage Control Characters

Character	Effect
Δ space	Advances one line
0 zero	Advances two lines
1 one	Advances to top of next page
+ plus	Does not advance (allows overprinting)
$ dollar sign	Advances one line before printing and suppresses carriage return at the end of the record

Any character other than those described in Table 6-2 is treated as though it were a space, and is deleted from the print line.

6.4 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 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)
```

6.5 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, an input field containing fewer than w characters may 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. It may be used in conjunction with I, O, F, E, D, G, and L field descriptors. For example:

```
        READ (6,100) I,J,A,B
    100 FORMAT (2I6,2F10.2)
```

If the external record input by the above statements contains:

```
1,-2,1.0,35
```

Then the following assignments take place:

I = 1

J = -2

A = 1.0

B = 0.35

Note that the physical end of the record also serves as a field terminator. Note also that the d part of a w.d specification is not affected as illustrated by the assignment to B.

Only fields of fewer than w characters may be terminated by a comma. If a field of w characters or greater is followed by a comma, the comma will be considered to be part of the following field.

Two successive commas, or a comma following a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor in question, the resulting value assigned is 0, 0.0, 0D0, or .FALSE..

A comma cannot be used to terminate a field that is to be read under control of an A, H, or alphanumeric literal field descriptor. If the physical end of the record is encountered before w characters have been read, however, short field termination is accomplished and the characters that were input are assigned successfully. Trailing spaces are appended to those characters to fill the associated I/O list element or the field descriptor.

6.6 OBJECT TIME FORMAT

Format specifications may be stored in 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:

```
        DOUBLEPRECISION FORRAY(6),RPAR,FBIG,FMED,FSML
        DATA FORRAY(1),RPAR,FBIG,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.

6.7 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. 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. An execution error occurs if this condition is not met.

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 Q 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 Q 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.

6.8 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.

6.8.1 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 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 in the range -127 to 127 inclusive. 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.

6.8.2 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 an integer constant. An external field input under control of an O field descriptor must have the form of an octal constant (Section 2.2.5), without a leading double quote. 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 (see Section 2.2.2). 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. 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.

6.8.3 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 line printer or terminal 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 7

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.

7.1 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
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
LOGICAL
LOGICAL*1
LOGICAL*4
```

[illegible]
LOGICAL*2

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 DOUBLE PRECISION D
IMPLICIT COMPLEX (S,Y), LOGICAL*1 (L,A-C)
```

7.2 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
LOGICAL*1
LOGICAL*4
INTEGER
INTEGER*2
INTEGER*4
REAL
```

```
REAL*4
DOUBLE PRECISION
REAL*8
COMPLEX
COMPLEX*8
```

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.

Also, a symbolic name may be followed by an optional length specification of the form *s, where s is one of the acceptable lengths for the data type being declared. Such a specification overrides, for the item with which it was specified, the length attribute implied by the statement. For example;

```
INTEGER*2 I,J,K,M12*4,Q,IVEC*4(10)
REAL*8 WX1,WXZ,WX3*4,WX5,WX6*8
```

Note that REAL*8 is the same as DOUBLE PRECISION.

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 be used to define names that are to represent any data type for which there is no implied definition, either by default or by specification in an IMPLICIT statement.

Type declaration statements must not be labeled.

Examples

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

7.3 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, four or eight bytes 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. The only exception to this rule is in the EQUIVALENCE statement, as described in Section 7.5.1.

DIMENSION statements must not be labeled.

Examples

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

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

DIMENSION MARK(4,4,4,4)
```

7.3.1 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.

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 20 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 may 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 dummy argument which is used in an array declarator must not change during the execution of the subprogram. For example, in the sample above, an assignment to the dummy argument N would be in error.

In addition, [illegible]

1. Each [illegible] is an [illegible] argument, or an [illegible] defined on entry to the subprogram.

2. Each operator is one of the [illegible] references and function [illegible]

Note that the upper and lower [illegible] time a subprogram is entered [illegible] of that subprogram even if the [illegible] array declaration are changed. For example, in

```
C MAIN PROGRAM
  DIMENSION ARRAY [illegible]
  L = 9
  M = 5
  CALL SUB(ARRAY,9,5)
  END

  SUBROUTINE SUB[illegible]
  DIMENSION X(-I/2[illegible])
  J = 1
  I = 2
  END
```

the adj[illegible] subro[illegible] declar[illegible]

7.4 COMMON STATEMENT

A COMMON statement defines one or more contiguous areas (blocks) of storage. 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.

7.4.1 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 be the same as a variable or array name; however, it may not be 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.

Care must be taken when LOGICAL*1 elements are assigned to common, to ensure that any data of other types, assigned following the LOGICAL*1 data, is allocated on a word boundary. All common blocks start on a word (even) boundary.

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.

7.4.2 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.

7.5 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 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*2  IARR(4)
EQUIVALENCE (DVAR,IARR(1))
```

The EQUIVALENCE statement causes the four 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 have the same definition provided they are also of the same data type.

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)
```

7.5.1 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 7-3.

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 7-3
Equivalence of Array Storage

Figure 7-3 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.

Equivalencing arrays with non-unity lower [illegible] fashion. For example, an array [illegible] values. A reference to A(2,2) [illegible] sequence. If array A(2:3,4) is to share [illegible] the following statement can be used.

```
EQUIVALENCE (A(8), B(10))
```

The entire array A occupies a [illegible] to array B as illustrated in [illegible] (A, B(3)) and EQUIVALENCE (A(10), [illegible] alignment of arrays.

Array B		Array A	
Array element	Element Number	[illegible]	[illegible]
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	[illegible]	[illegible]
B(2,2)	4	[illegible]	[illegible]
B(3,2)	5	[illegible]	[illegible]
B(4,2)	6	A(3,2)	[illegible]
B(2,3)	7	A(2,3)	[illegible]
B(3,3)	8	A(3,3)	[illegible]
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Figure 7-4
Equivalence of Arrays with Non-Unity Lower [illegible]

7.5.2 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)
Existing Common				Extended Portion		

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)
Extended Portion	Existing Common				Extended Portion

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

7.5.3 EQUIVALENCE and LOGICAL*1 Arrays

If an element of a LOGICAL*1 array that is not aligned on a word boundary is equivalenced to an array or variable of another data type, it may cause that variable or all elements of that array not to be aligned on word boundaries. If this occurs, an attempt to reference that variable or those array elements will cause an error during execution of the program.

7.6 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 argrument to a subprogram. The subprogram may then use the associated dummy argument name in a function reference or a CALL 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; the function name need not be defined in an EXTERNAL statement.

Example

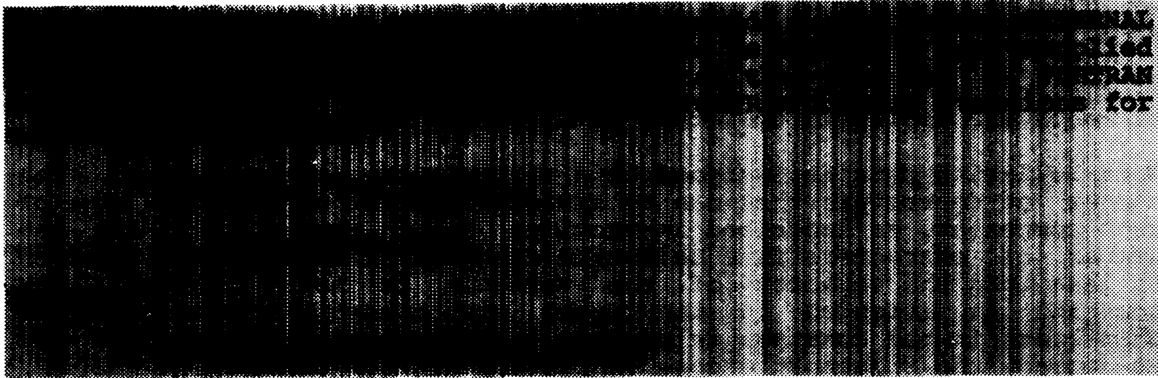
```
     Main Program                        Subprograms

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.)



7.7 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 constant.

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.

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, as described in Sections 2.3.3 and 2.3.4. If the number of characters in a Hollerith constant or alphanumeric literal 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.

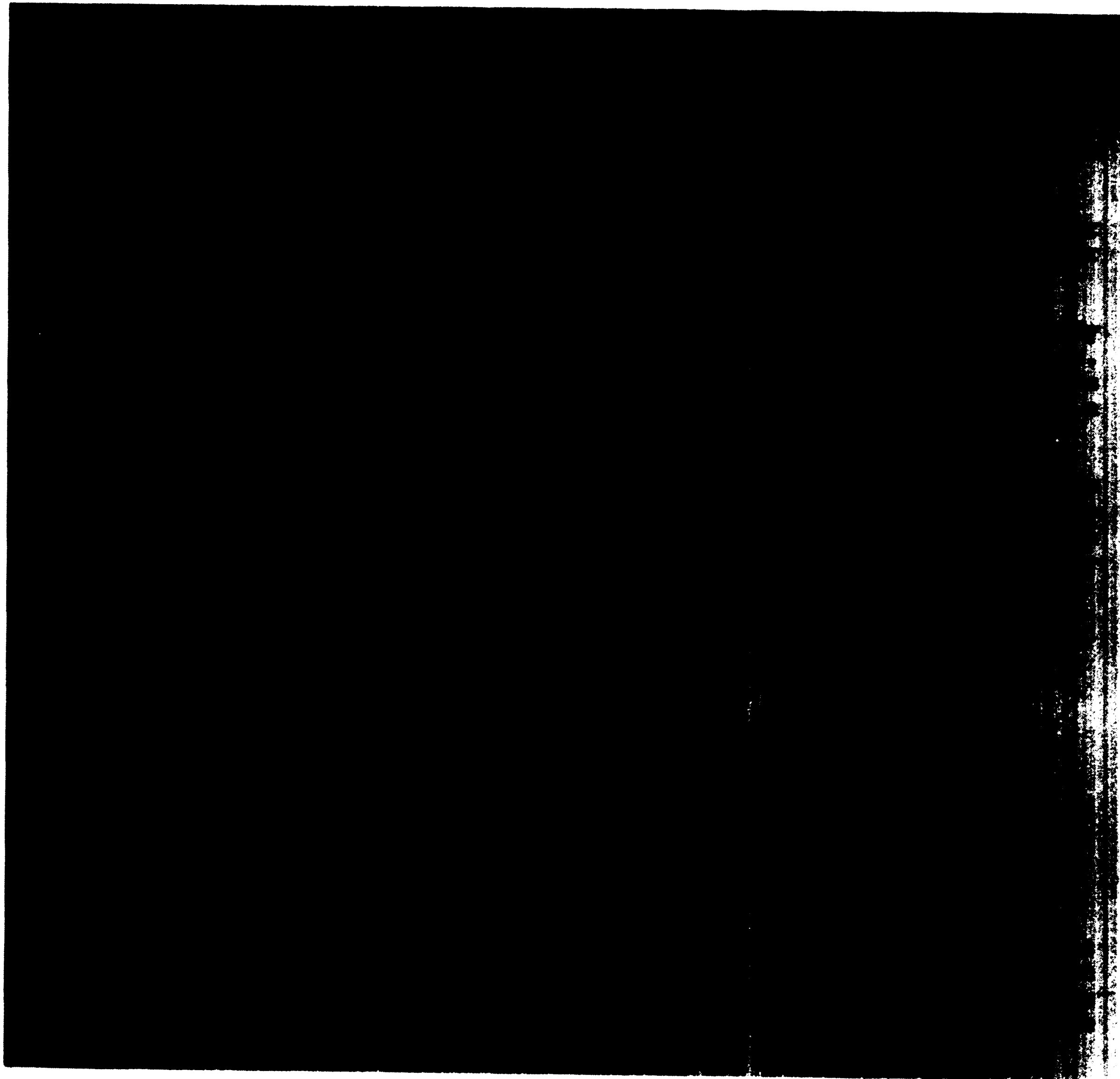
The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names should also agree (except in the case of alphanumeric data).

Radix-50 constants may be used to initialize Real variables only.

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 to the real variable STARS.



CHAPTER 8

SUBPROGRAMS

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, and subroutines.

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 always returns a value to the calling program. 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 appears in the following general form:

```
f ([p[,p]...])=e
```

f is a symbolic name.

p is a symbolic name.

e is an expression.

f is the name of the ASF, in the same form as a variable name.

Each p is a dummy argument, which must be a valid variable name. The expression is an arithmetic expression that defines the computation to be performed by the ASF.

A function reference to an ASF 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.

Dummy arguments in an ASF definition serve only to indicate the number, order, and data type of the actual arguments. The same names may be used to represent other entities elsewhere in the program unit. 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.

The expression in an ASF definition may contain function references. If a reference to another ASF appears in the expression, that function must have been defined previously. The definition of an arithmetic statement function must not contain a reference to itself.

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[*n][([p[,p]...])]
```

typ is a type specifier.

nam is a symbolic name.

*n is a type override.

p is a symbolic name.

nam is the symbolic name of the function, in the same form as a variable name; typ is the name of a data type, *n is optional and is any legal length specifier for the type of the function, and each p is an optional 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.

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.

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 or expression, the function must not attempt to alter the value of the corresponding dummy argument.

A FUNCTION subprogram may contain references to other subprograms, but 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.

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.*1E-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:

$$X_{i+1} = X_i - \frac{\cosh(X_i)+\cos(X_i)-A}{\sinh(X_i)-\sin(X_i)}$$

which is repeatedly calculated until the difference between X_i and X_{i+1} is less than 1×10^{-6}. The function makes use of the FORTRAN Library functions EXP, SIN, COS, and ABS.

8.1.3 SUBROUTINE Subprogram

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 body of the SUBROUTINE subprogram is similar to any other program unit except that it must contain at least one RETURN statement.

The form of the CALL statement is described in Section 4.5.

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 any number of 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.

Statements in a subroutine may establish or redefine values for any argument whose associated actual argument is not a constant, expression or subprogram name.

Actual arguments in a CALL statement must agree in number, order, and data type with the dummy arguments in the corresponding SUBROUTINE statement.

Example

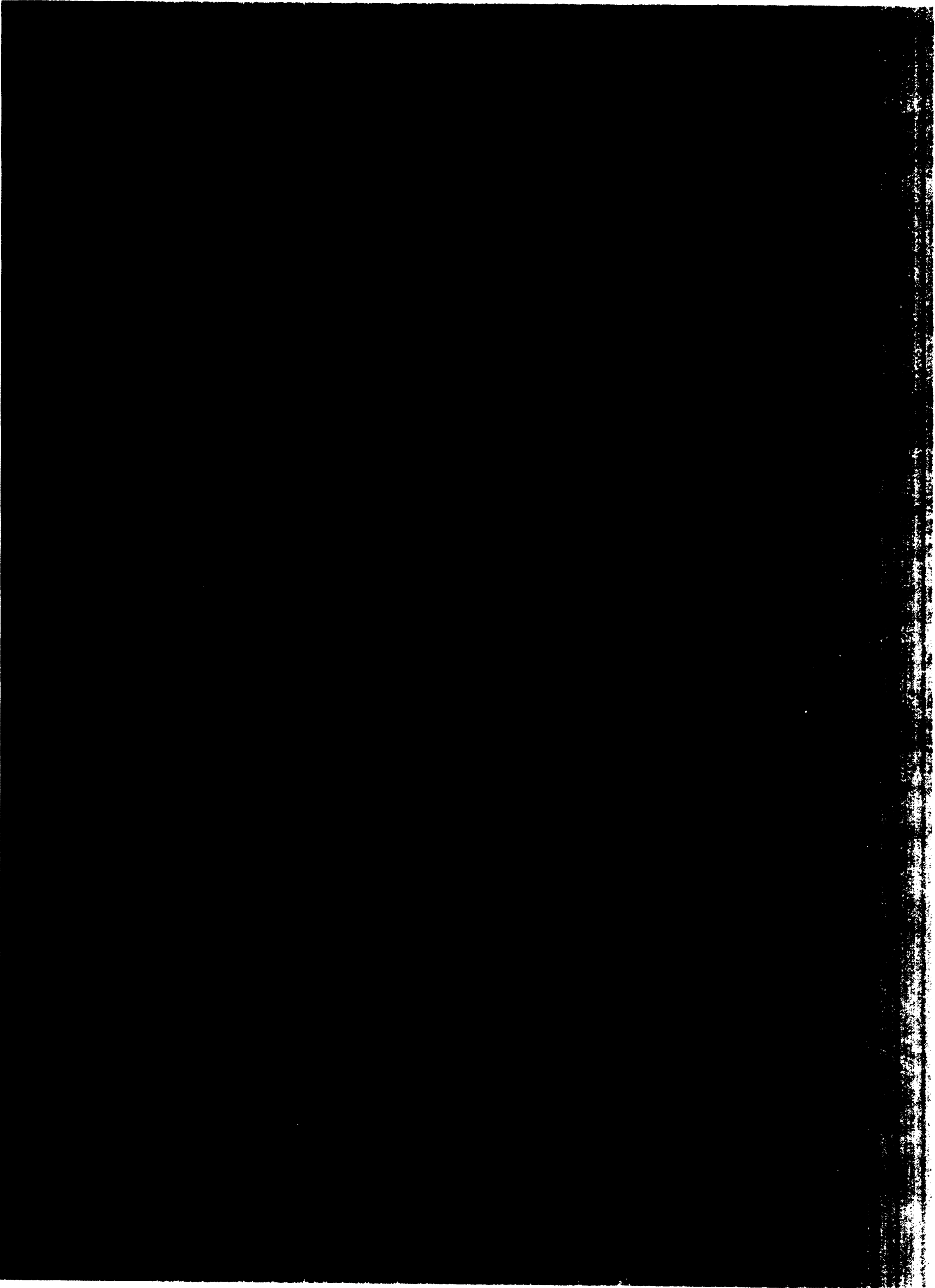
Main Program

```
C       .
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
```

SUBROUTINE Subprogram

```
        SUBROUTINE PLYVOL
        COMMON NFACES,EDGE,VOLUME
        CUBED = EDGE**3
        GO TO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5,6), 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 REGULAR 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 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.



8.1.5 BLOCK DATA Subprogram

The BLOCK DATA subprogram is used to assign initial values to entities in labeled common blocks, at the same time establishing and defining those blocks. 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.

In FORTRAN [illegible], the additional [illegible]

BLOCKDATA nam

nam is a symbolic name.

The BLOCK DATA subprogram functions at compilation time only. The specification statements in the subprogram establish and define common blocks, assign variables and arrays to those blocks, 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. Initial values may be entered into more than one block by the same subprogram.

Example

```
BLOCK DATA
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/AREA2/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 8-1. 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,

```
R = 3.14159 * ABS(X-1)
```

causes the absolute value of X-1 to be calculated, multiplied by the constant 3.14159, and assigned to the variable R.

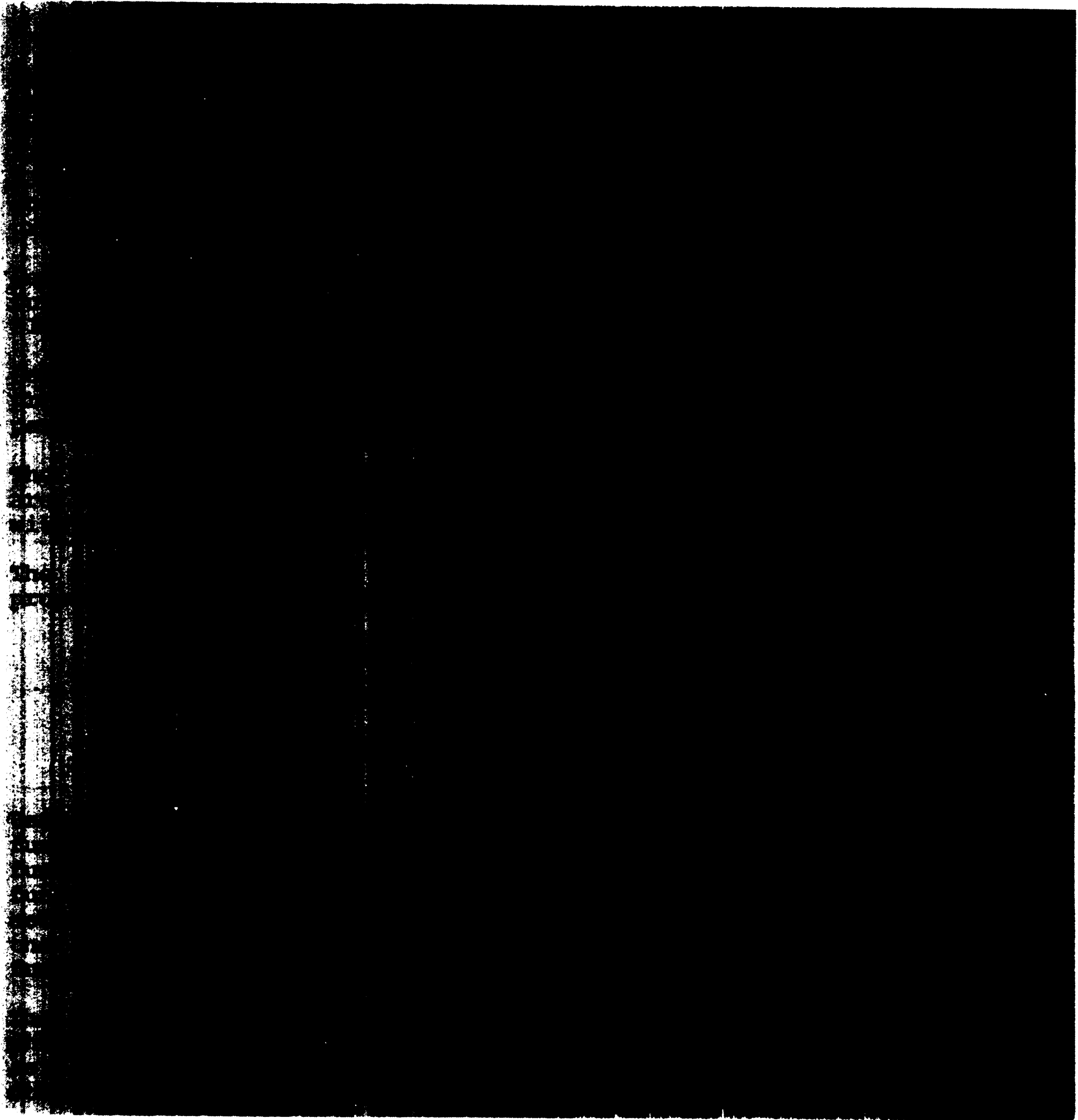
The data type of each library function is predefined as described in Table 8-1. Arguments passed to these functions may consist of subscripted or simple variable names, expressions, constants, arithmetic functions. Arguments to these functions must correspond to the type indicated in Table 8-1.

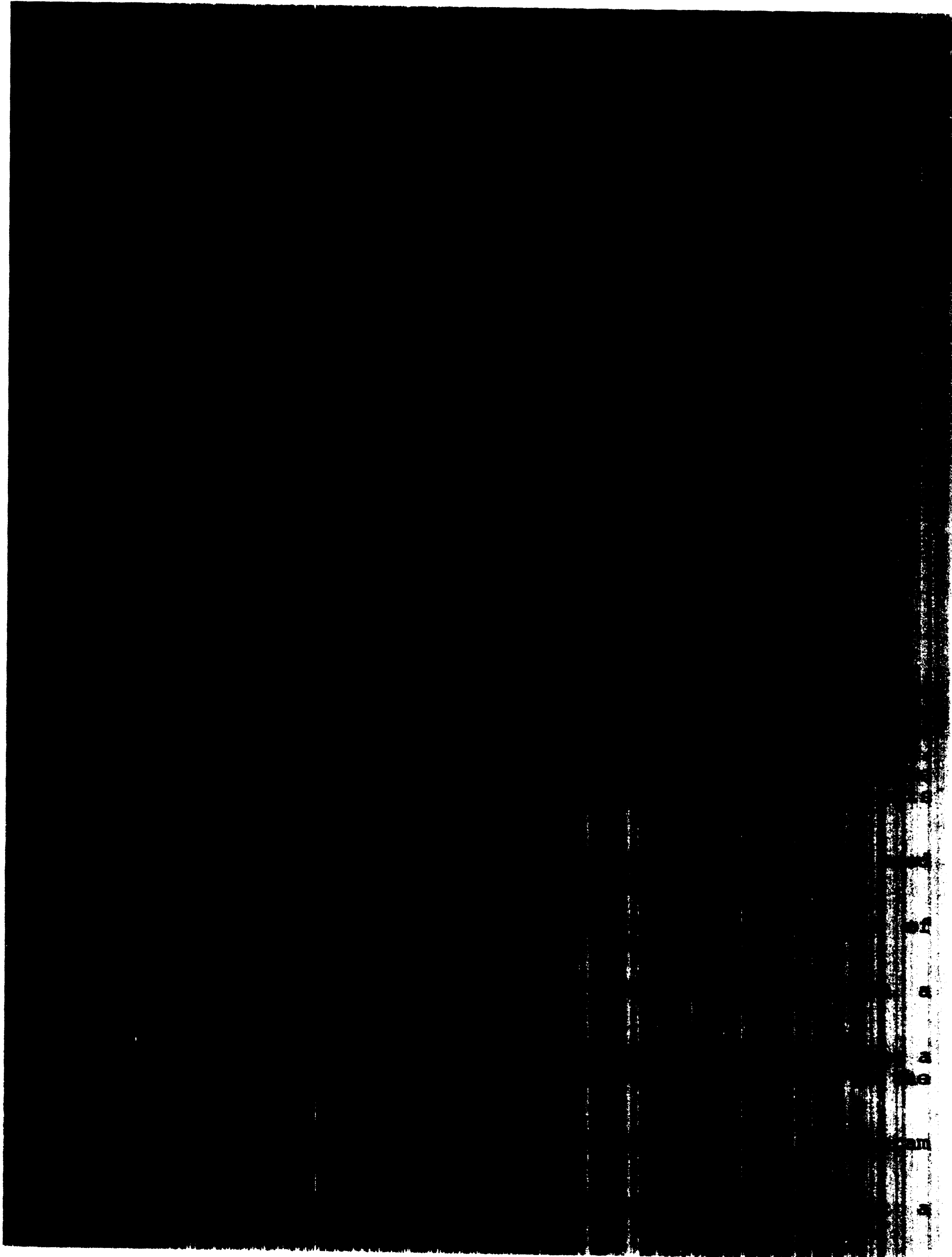
Table 8-1
FORTRAN Library Functions

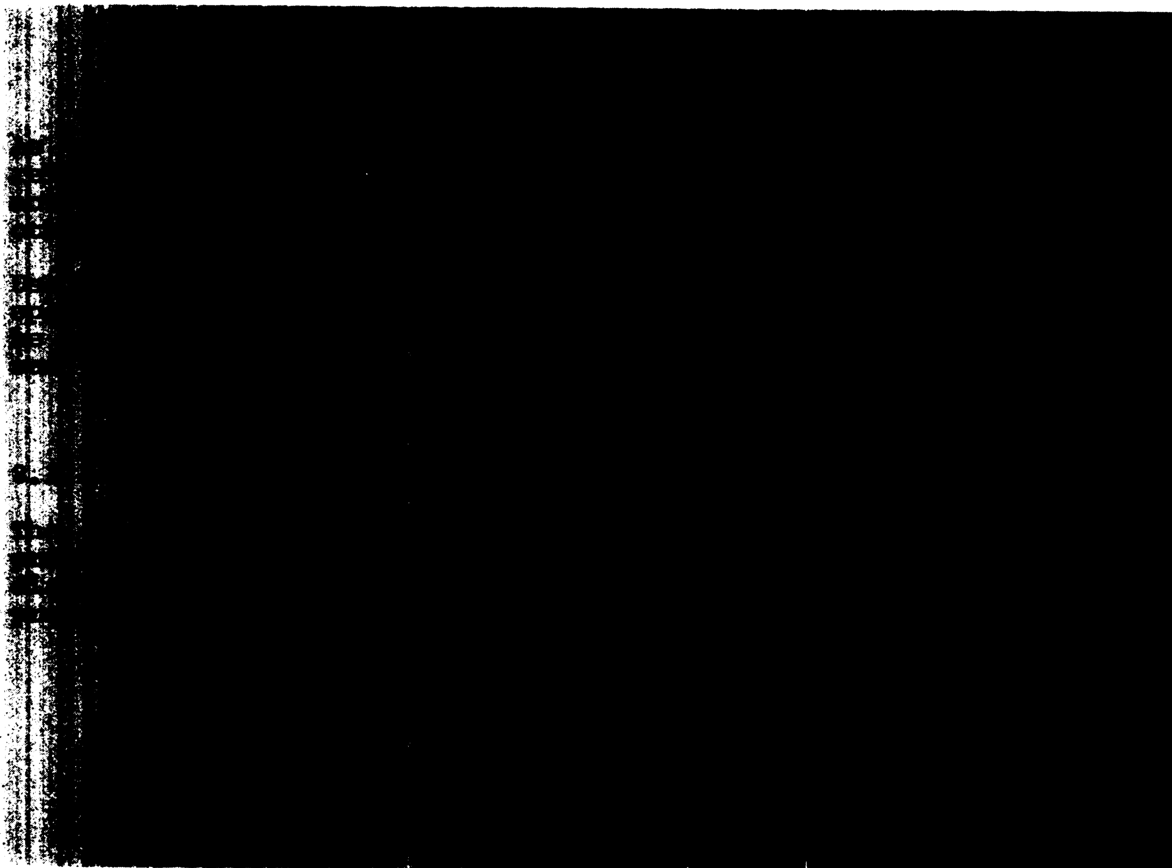
FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value where Z=(x,y) CABS(Z)=$(x^2+y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion, obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion CMPLX(X,Y)=X+i*Y	Real	Complex
	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 to Integer truncation	Double	Integer
	Remainder functions return the remainder when the first argument is divided by the second.		
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
	Maximum value functions return the largest value from among the argument list; ≥ 2 arguments.		
AMAXØ(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAXØ(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double
	Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments.		
AMINØ(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MINØ(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	Double

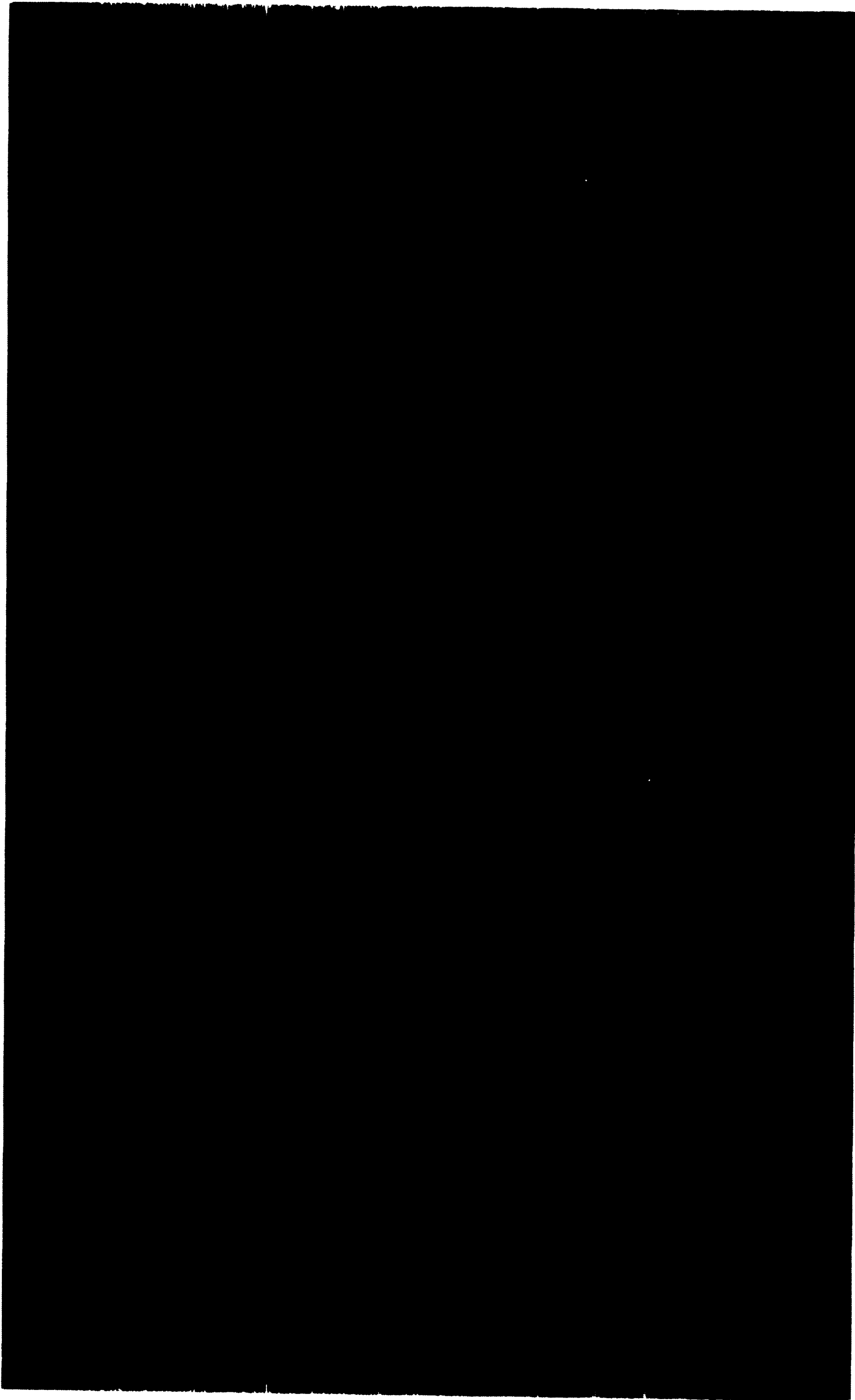
Table 8-1 (Cont.)
FORTRAN Library Functions

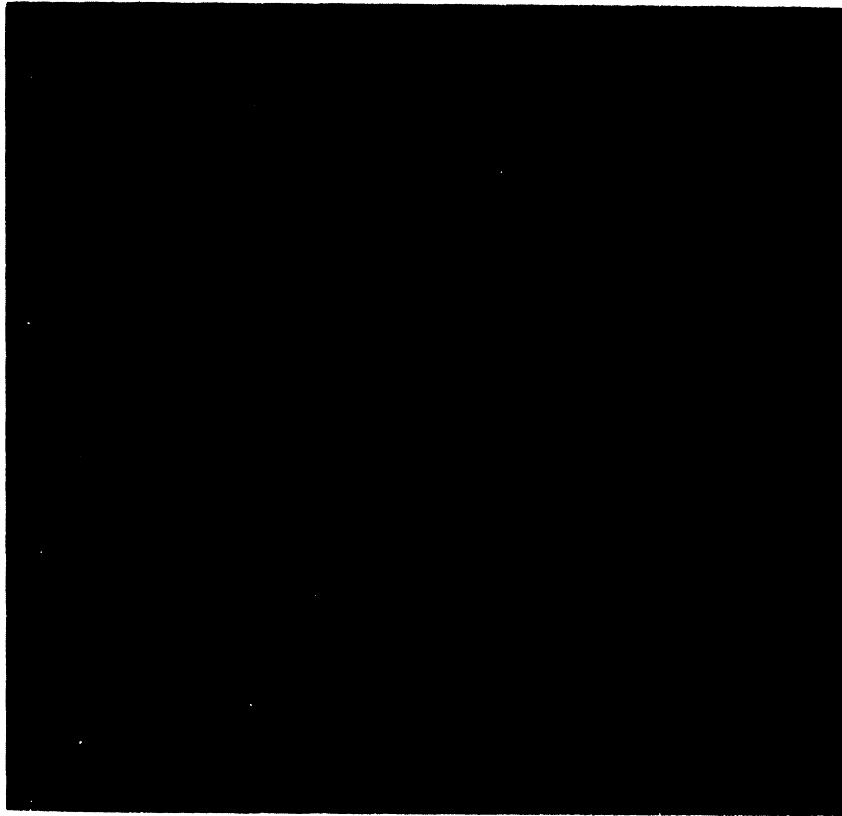
FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
	The transfer of sign functions return (sign of the second argument) * (absolute value of first argument).		
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(X,Y)	Double precision transfer of sign	Double	Double
	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
	Exponential functions return the value of e raised to the argument power.		
EXP(X)	e^x	Real	Real
DEXP(X)	e^x	Double	Double
CEXP(Z)	e^z	Complex	Complex
ALOG(X)	Returns $\log_e$(X)	Real	Real
ALOG1Ø(X)	Returns $\log_{1Ø}$(X)	Real	Real
DLOG(X)	Returns $\log_e$(X)	Double	Double
DLOG1Ø(X)	Returns $\log_{1Ø}$(X)	Double	Double
CLOG(Z)	Returns $\log_e$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if Z=X+i*Y COMJG(Z)=Z-i+y	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range Ø to 1. I and J must be integer variables and should be set initially to Ø. Resetting I and J to Ø regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences. See also Appendix C.3.	Integer	Real

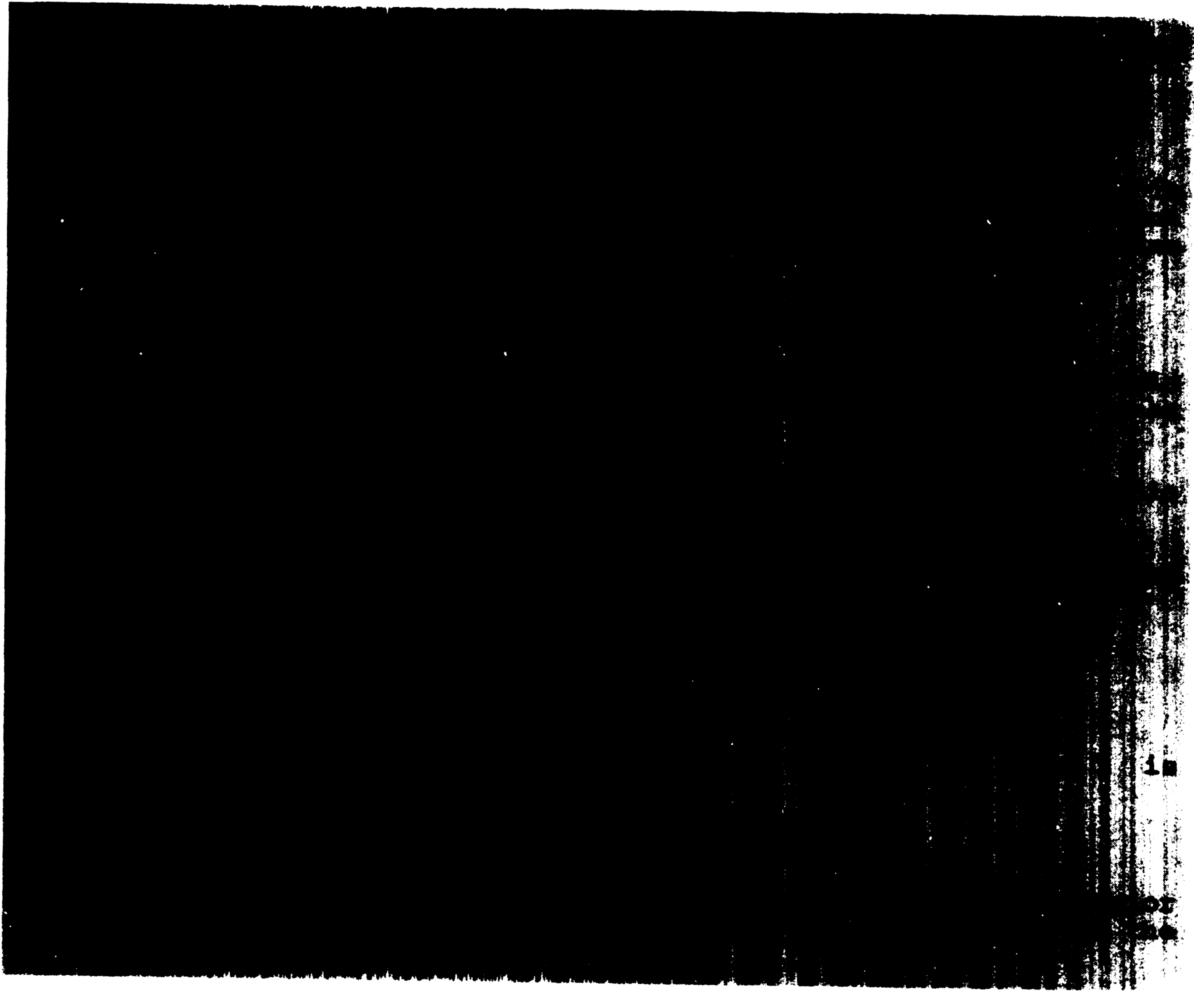












APPENDIX A

CHARACTER CODES

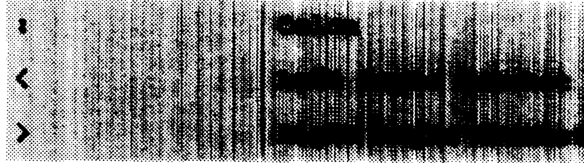
A.1 FORTRAN CHARACTER SET

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
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe
"	Double Quote
$	Dollar Sign

APPENDIX A



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

A.2 ASCII CHARACTER CODE

CHARACTER	PARITY ASCII	DECØ29	DECØ26	CHARACTER	PARITY ASCII	DECØ29	DECØ26
{	173	12 Ø	12 Ø				
}	175	11 Ø	11 Ø				
SPACE	24Ø	NONE	NONE	@	3ØØ	8 4	8 4
!	Ø41	12 8 7	12 8 7	A	3Ø1	12 1	12 1
"	Ø42	8 7	Ø 8 5	B	3Ø2	12 2	12 2
#	243	8 3	Ø 8 6	C	3Ø3	12 3	12 3
$	Ø44	11 8 3	11 8 3	D	1Ø4	12 4	12 4
%	245	Ø 8 4	Ø 8 7	E	3Ø5	12 5	12 5
&	246	12	11 8 7	F	3Ø6	12 6	12 6
'	Ø47	8 5	8 6	G	3Ø7	12 7	12 7
(	Ø5Ø	12 8 5	Ø 8 4	H	31Ø	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
+	Ø53	12 8 6	12	K	313	11 2	11 2
,	254	Ø 8 3	Ø 8 3	L	314	11 3	11 3
-	Ø55	11	11	M	315	11 4	11 4
.	Ø56	12 8 3	12 8 3	N	316	11 5	11 5
/	257	Ø 1	Ø 1	O	317	11 6	11 6
Ø	Ø6Ø	Ø	Ø	P	32Ø	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	Ø63	3	3	S	323	Ø 2	Ø 2
4	264	4	4	T	324	Ø 3	Ø 3
5	Ø65	5	5	U	325	Ø 4	Ø 4
6	Ø66	6	6	V	326	Ø 5	Ø 5
7	267	7	7	W	327	Ø 6	Ø 6
8	27Ø	8	8	X	33Ø	Ø 7	Ø 7
9	Ø71	9	9	Y	331	Ø 8	Ø 8
:	Ø72	8 2	11 8 2	Z	332	Ø 9	Ø 9
;	273	11 8 6	Ø 8 2	[	333	12 8 2	11 8 5
<	Ø74	12 8 4	12 8 6	\	334	Ø 8 2	8 7
=	275	8 6	8 3	]	335	11 8 2	12 8 5
>	276	Ø 8 6	11 8 6	↑ or ^	336	11 8 7	8 5
?	Ø77	Ø 8 7	12 8 2	← or _	337	Ø 8 5	8 2

A.3 RADIX-50 CHARACTER CODE

Radix-50 is a special character data representation in which up to three characters from the Radix-50 character set (a subset of the ASCII character set) can be encoded and packed into a single PDP-11 storage word.

The Radix-50 characters and their corresponding code values are as follows:

Character	ASCII Octal Equivalent	Radix-50 Value (Octal)
Space	40	0
A - Z	101 - 132	1 - 32
$	44	33
.	56	34
(Unassigned)		35
0 - 9	60 - 71	36 - 47

Radix-50 values are stored, up to three characters per word, by packing them into single numeric values according to the formula:

((i * 50 + j) * 50 + k)

where "i", "j", and "k" represent the code values of three Radix-50 characters.

The maximum Radix-50 value is, thus,

47*50*50 + 47*50 + 47 = 174777

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic is performed in octal):

```
  X = 113000
  2 = 002400
  B = 000002
 ------------
X2B = 115402
```

Radix-50 Character/Position Table

Single Char. or First Char.		Second Character		Third Character	
Blank	000000	Blank	000000	Blank	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
UNUSED	132500	UNUSED	002210	UNUSED	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

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, and expressions
	*,/	multiplication, division	
	+,-	addition, subtraction unary plus and minus	
Relational	.GT.	greater than	arithmetic or logical constants, variables, 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, 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	
	.EQV.	A.EQV.B is true if and only if A and B are both true or A and B are both false.	(precedence same as .XOR.)
	.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.	(precedence same as .EQV.)

B.2 STATEMENTS

The following summary of statements available in the PDP-11 FORTRAN language defines the general format for the statement. If more detailed information is needed, the reader is referred to the Section(s) in the manual dealing with that particular statement.

Statement Formats	Effect	Manual Section
ACCEPT	See READ, formatted	5.4.3
Arithmetic/Logical Assignment		
v=e		3.1
v	is a variable name or an array element name	
e	is an expression	
	The value of the arithmetic or logical expression is assigned to the variable.	
Arithmetic Statement Function		
f([p[,p]...])=e		8.1.1
f	is a symbolic name	
p	is a symbolic name	
e	is an 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 — 5.8.2

u — is an integer variable or constant

The currently open file on logical unit number u is backspaced one record.

BLOCK DATA — 8.1.5

[illegible]

nam — is a symbolic name

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.

CLOSE (p[,p]...) — 5.8.6

p — is one of the following forms:

key	a keyword
key=e	e is an integer expression
key=s	s is an executable statement label
key=lit	lit is an alphanumeric literal
key=v	v is an integer variable name
key=n	n is an array name, variable name, array element name, or alphanumeric literal

Closes the specified file.

COMMON [/[cb]/] nlist [/[cb]/ nlist]... 7.4

COMMON [/[cb]/] nlist [[,]/[cb]/nlist]...

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.

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

DATA [illegible] 7.7

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.

DECODE (c,f,b)[list] 5.9

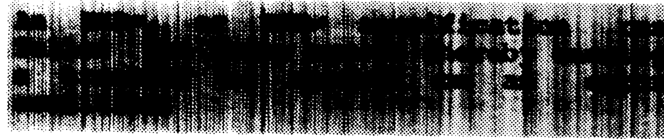
c is an integer expression

f is a FORMAT statement label or array name

b is a variable name, an array name, or an array element name

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 b is the name of an array containing the ASCII characters to be converted.



DEFINE FILE u(m,n,U,v)[,u(m,n,U,v)]... 5.5.3

u   is an integer variable name or integer constant

m   is an integer variable name or integer constant

n   is an integer variable name or integer constant

v   is an integer variable name

Defines the record structure of a direct access file where u is the logical unit number, m is the number of fixed length records in the file, n is the length in words of a single record, U is a fixed argument, and v is the associated variable.

DIMENSION a(d)[,a(d)]... 7.3

a(d)   is an array declarator

Specifies storage space requirements for arrays.

DO s i = e1,e2[,e3] 4.3

s   is the label of an executable statement

i   is a variable name

ei   are integer expressions

1. Set i = e1

2. Execute statements through statement number s

3. Evaluate i = i+e3

4. Repeat 2 through 3 for
   MAX (1, INT((e2 - e1)/e3) + 1)
   iterations

ENCODE (c,f,b)[list] 5.9

c is an integer expression

f is a FORMAT statement label or an array name

b is a variable name, array name, or array element name

list is an I/O list

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 b is the name of the array to be used as a buffer.

END 4.9

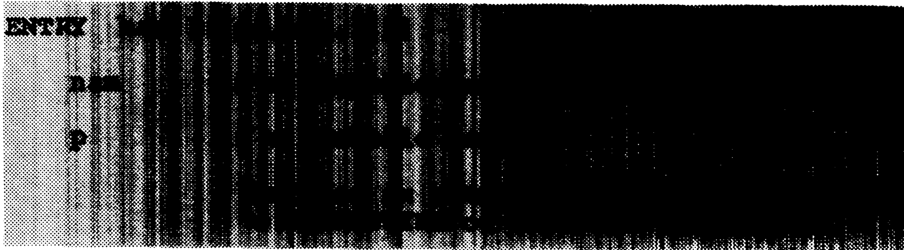
Delimits a program unit.

END FILE u 5.8.3

u is an integer variable or constant

An end-file record is written on logical unit u.

8.1.4



EQUIVALENCE (nlist)[,(nlist)]... 7.5

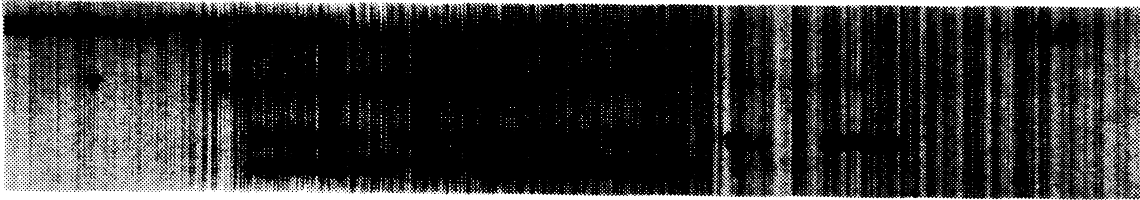
nlist is a list of two or more variable names, array names, or array element names separated by commas. Subscript expressions must be constant.

Each of the names (nlist) within a set of parentheses is assigned the same storage location.

EXTERNAL v[,v]... 7.6

v is a procedure name

Informs the system that the names specified are those of FUNCTION or SUBROUTINE programs.



FIND (u'r) 5.8.4

u is an integer variable name or integer constant

r is an integer expression

Positions the file on logical unit number u to record r and sets associated variable to record number r.

FORMAT (field specification,...) 6.1 - 6.8

Describes the format in which one or more records are to be transmitted; a statement label must be present.

[typ] FUNCTION nam[*n][([p[,p]...])]   **8.1.2**

- typ — is a type specifier
- nam — is a symbolic name
- *n — is a type override
- 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)[,] e   **4.1.2**

- slist — is a list of one or more executable statement labels separated by commas.
- e — is an integer expression

(Computed GO TO) Transfers control to the statement label specified by the value of expression e. (If e=1 control transfers to the first statement label. If e=2 it transfers to the second statement label. etc.) If e is less than 1 or greater than the number of statement labels present, no transfer takes place.

GO TO v[,(slist)]

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.

IF (e) s1,s2,s3 — 4.2.1

e — is an expression

si — are executable statement labels

(Arithmetic IF) Transfers control to statement number si depending upon the value of the expression. If the value of the expression is less than zero, transfer to s1; if the value of the expression is equal to zero, transfer to s2; if the value of the expression is greater than zero, transfer to s3.

IF (e) st — 4.2.2

e — is an expression

st — is any executable statement except a DO or a logical IF statement

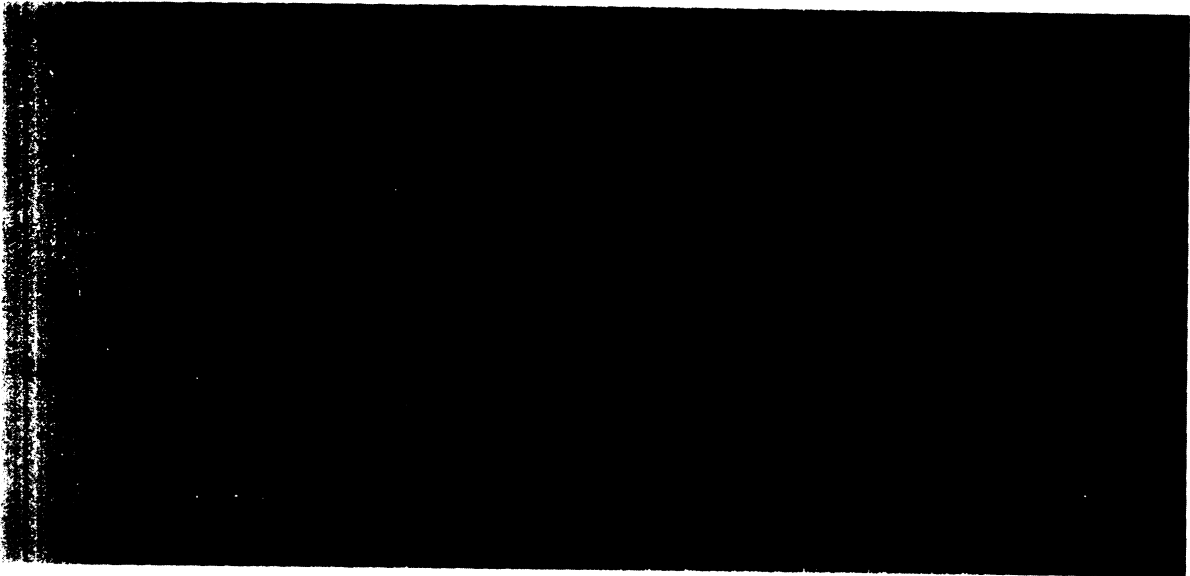
(Logical IF) Executes the statement if the logical expression is true.

IMPLICIT typ (a[,a]...)[,typ(a[,a]...)]... — 7.1

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 a decimal digit string containing one to five digits, an alphanumeric literal, or an octal constant

Suspends program execution and prints the display, if one is specified.

PRINT See WRITE, formatted 5.4.5

READ f[,list] 5.4.1

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

ACCEPT f[,list]

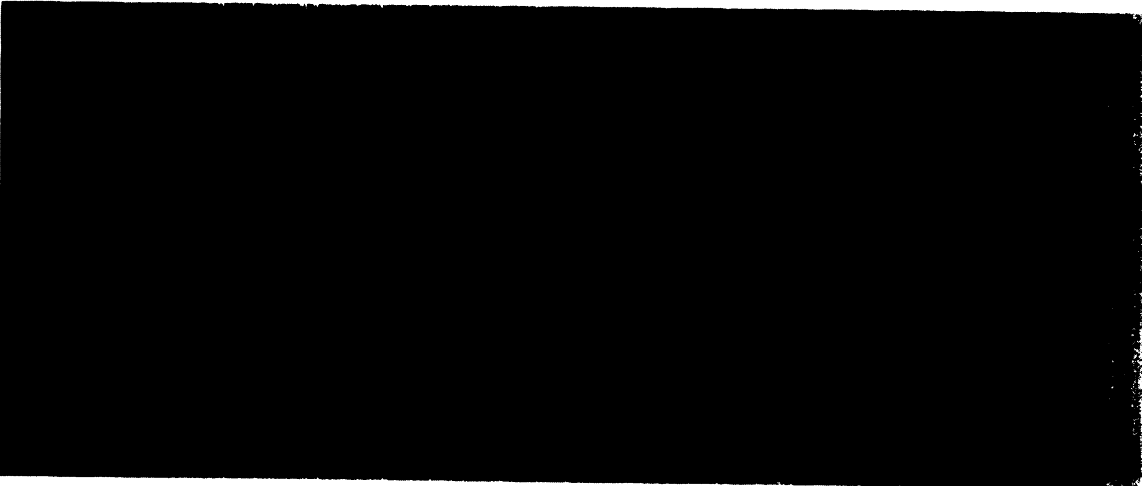
u is an integer variable or constant

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[,END=s][,ERR=s])[list] 5.3.1

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] 5.5.1

u is an integer variable or constant

r is an integer expression

s is an executable statement label

list is an I/O list

(Unformatted Direct Access READ) Reads record r from logical unit u, assigning values to the variables in the optional list.

RETURN 4.6

Returns control to the calling program from the current subprogram.

REWIND u 5.8.1

u is an integer variable or constant

Repositions logical unit number u to the beginning of the currently opened file.

STOP [disp] 4.8

disp is a decimal digit string containing one to five digits, an alphanumeric literal, or an octal constant

Terminate program execution and print 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, formatted 5.4.4

Type Declaration

typ v[,v]... 7.2

typ is a data type specifier

v is a variable name, an array name, a function or function entry name, or an array declarator. The name can optionally be followed by a length modifier (*n).

(Type Declarations) The symbolic names, v, are assigned the specified data type in the program unit.

Typ is one of:

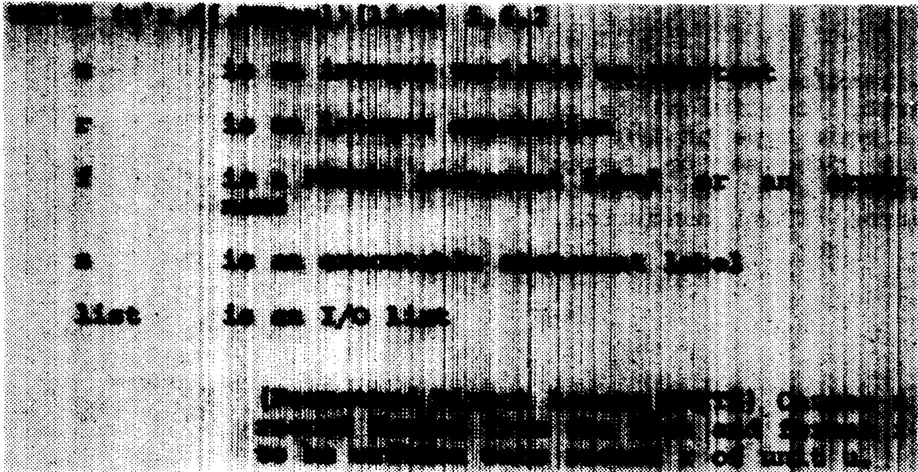
DOUBLE PRECISION
COMPLEX
COMPLEX*8
REAL
REAL*4
REAL*8

INTEGER
INTEGER*2
INTEGER*4
LOGICAL
LOGICAL*1
LOGICAL*4

WRITE (u,f[,ERR=s])[list] **5.4.2**
PRINT f[,list]
TYPE f[,list]

- 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[,ERR=s])[list] — 5.3.2

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.

WRITE (u'r[,ERR=s]) [list] — 5.5.2

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.

END=s,ERR=s — 5.7

(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.

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	Double
CABS(Z)	Complex to Real, absolute value where Z=(x,y) CABS(Z)=$(x^2+y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion, obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion CMPLX(X,Y)=X+i*Y	Real	Complex
	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 to Integer truncation	Double	Integer
	Remainder functions return the remainder when the first argument is divided by the second.		
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
	Maximum value functions return the largest value from among the argument list; ≥ 2 arguments.		
AMAXØ(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAXØ(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double
	Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments.		
AMINØ(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MINØ(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	Double

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 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	Double
	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
	Exponential functions return the value of e raised to the argument power.		
EXP(X)	e^x	Real	Real
DEXP(X)	e^x	Double	Double
CEXP(Z)	e^z	Complex	Complex
ALOG(X)	Returns $\log_e$(X)	Real	Real
ALOG1Ø(X)	Returns $\log_{1Ø}$(X)	Real	Real
DLOG(X)	Returns $\log_e$(X)	Double	Double
DLOG1Ø(X)	Returns $\log_{1Ø}$(X)	Double	Double
CLOG(Z)	Returns $\log_e$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if Z=X+i*Y COMJG(Z)=Z-i+y	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range Ø to 1. I and J must be integer variables and should be set initially to Ø. Resetting I and J to Ø regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences. See also Appendix C.3.	Integer	Real

$ABS
$DABS
$CABS
$IABS
$JABS

$IABS
$JABS

$COS
$DCOS
$CCOS

$EXP
$DEXP
$CEXP

$ALOG
$DLOG
$CLOG

$ALG1Ø
$DLG1Ø

$SIN
$DSIN
$CSIN

$SQRT
$DSQRT
$CSQRT

$TAN
$DTAN

$ACOS
$DACOS

$ASIN
$DASIN

$ATAN
$DATAN

$ATAN2
$DATN2

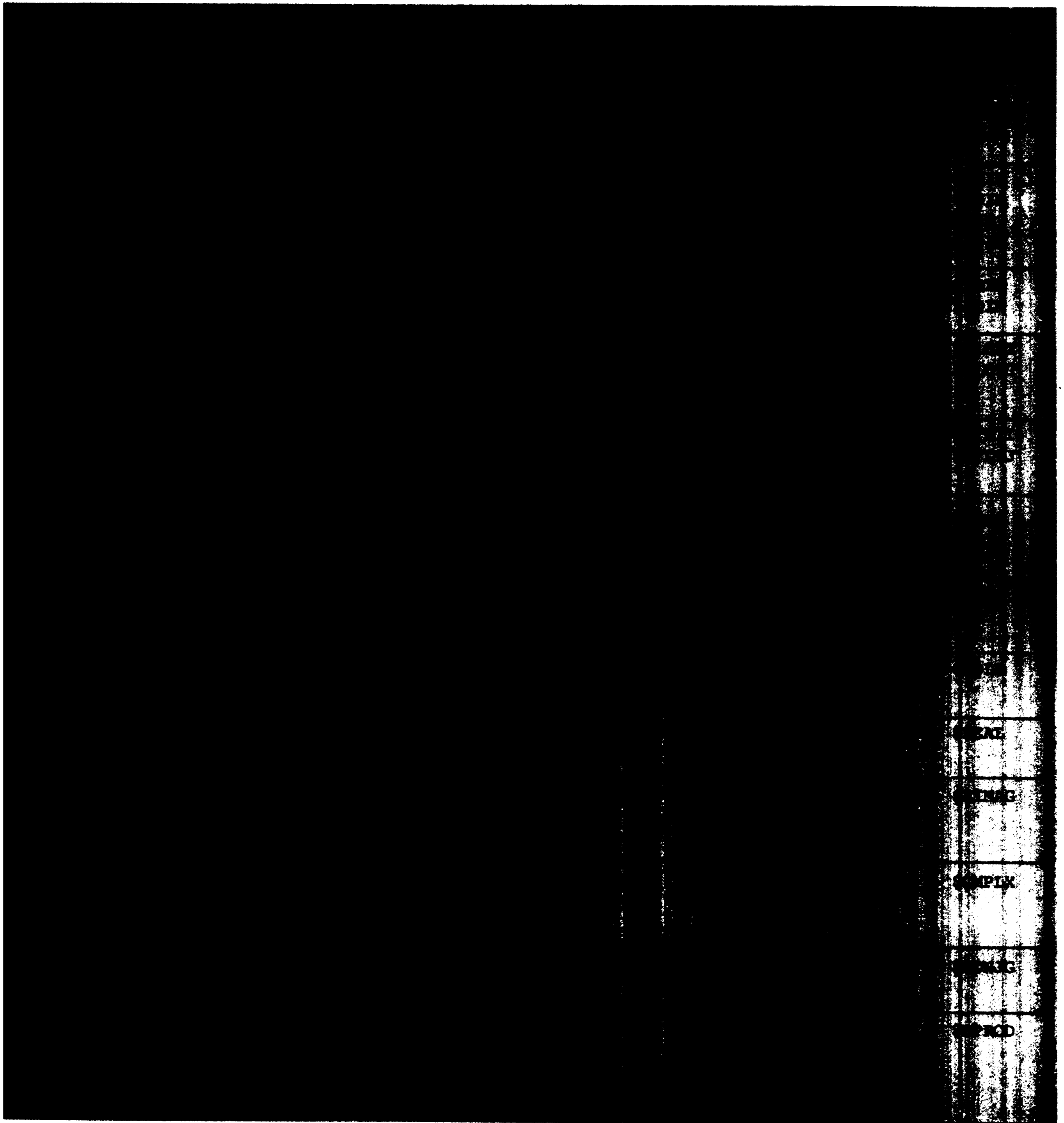
$COSH
$DCOSH

$SINH
$DSINH

$TANH
$DTANH

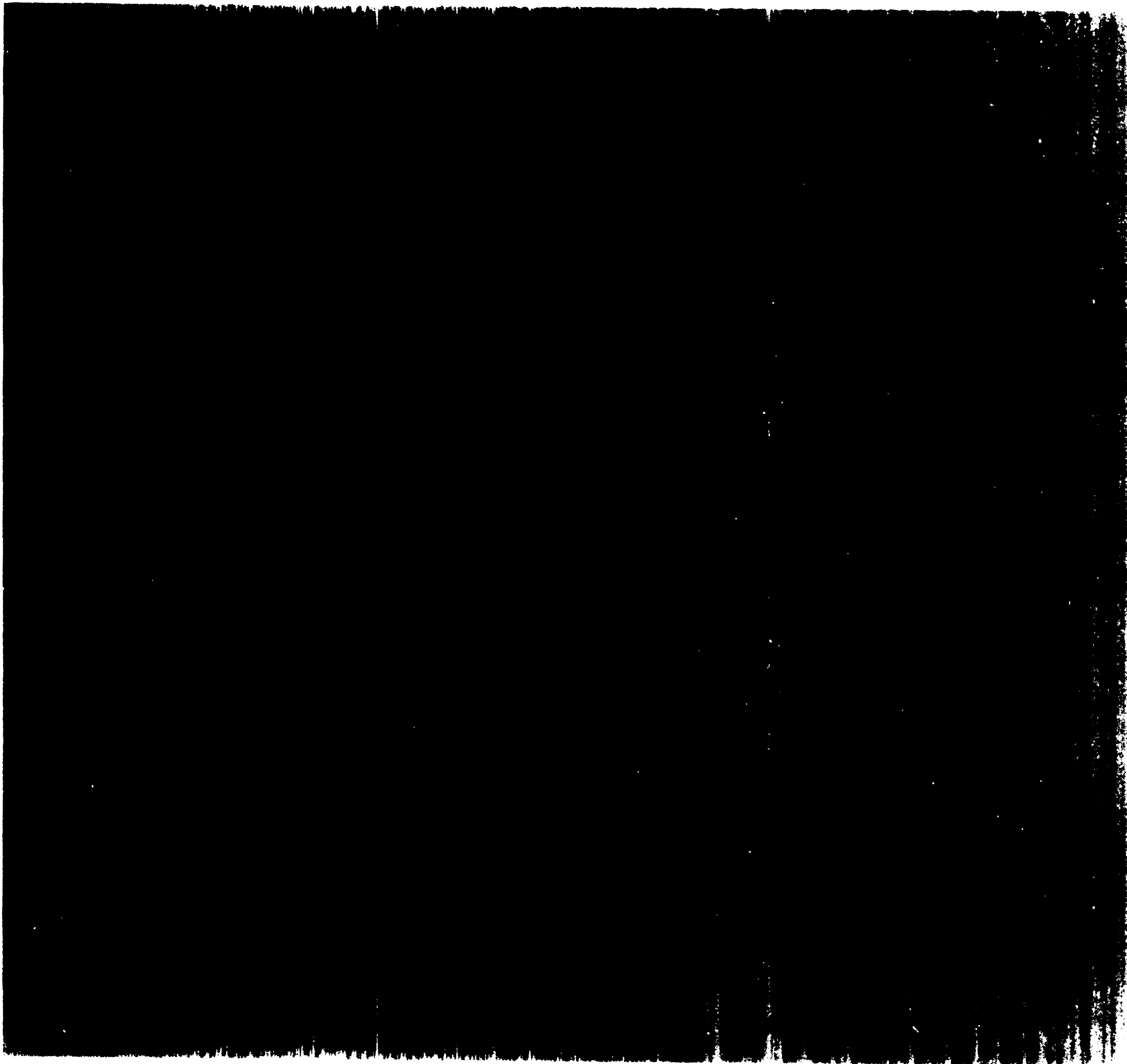
Function
Maximum
Minimum
Truncation
Remainder
Transfer of Sign

* = Result Generic



* = Result Generic

Function
Bitwise AND
Bitwise Inclusive OR
Bitwise Exclusive OR
Bitwise Compl[illegible]
Bitwise Shift



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:

```
        TYPE 5
5       FORMAT (//' THIS PROGRAM PERFORMS LINEAR REGRESSION'/
        1 ' THE LINE WHICH BEST FITS A SET OF X,Y PAIRS IS CALCULATED'/)
10      TYPE 20
20      FORMAT (/' TYPE IN THE NUMBER OF X,Y PAIRS: ',$)
        ACCEPT 50,N
50      FORMAT (I2)
        IF (N .LE. 0) STOP
        TYPE 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
                ACCEPT 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 (6,300) 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
```

Example four demonstrates a simple way to generate random numbers in a given range using the FORTRAN Library function RAN. A program listing and sample run follow:

EXAMPLE 4 LISTING:

```
        REAL MAX,MIN
        TYPE 10
10      FORMAT(1X,'THIS IS A PROGRAM TO GENERATE A FILE OF',
        1  ' RANDOM #''S IN A GIVEN RANGE.'/)
        TYPE 20
20      FORMAT(1X,'ENTER THE NUMBER OF RANDOM #''S TO GENERATE: ',$)
        ACCEPT 30,J
30      FORMAT(I3)
        TYPE 40
40      FORMAT(1X,'ENTER THE MINIMUM VALUE ',$)
        ACCEPT 60, MIN
        TYPE 50
50      FORMAT(1X,'ENTER THE MAXIMIM VALUE ',$)
        ACCEPT 60, MAX
60      FORMAT(F10.4)
        L = 0
        M = 0
        DO 100 K=1,J
                X = RAN(L,M) * (MAX-MIN) + MIN
                WRITE(1,60) X
100     CONTINUE
        REWIND 1
        DO 101 K=1,J
                READ (1,60) X
                TYPE 60,X
101     CONTINUE
        STOP
        END
```

EXAMPLE 4 SAMPLE RUN:

```
THIS IS A PROGRAM TO GENERATE A FILE OF RANDOM #'S IN A GIVEN RANGE.

ENTER THE NUMBER OF RANDOM #'S TO GENERATE. 20
ENTER THE MINIMUM VALUE 11.
ENTER THE MAXIMIM VALUE 36.
  11.0008
  11.0046
  11.0206
  11.0824
  11.3090
  12.1124
  14.8933
  24.3485
  31.0510
  11.1701
  31.5610
  32.8354
  31.9635
  15.2625
  22.9033
  19.0573
  27.2136
  35.7662
  13.6746
  29.1519
```

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:

```
          LOGICAL*1 STARS(80)
          INTEGER ARRAY(200),HIST(10)
          DATA STARS/80*'*'/
          DO 10 I=1,200
                  READ(1,20,END=100) ARRAY(I)
20                FORMAT (I3)
10        CONTINUE
100       ISIZE = I-1
          DO 120 J=1,ISIZE-1
                  DO 110 K=J+1,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
140               FORMAT (/1X,I3,' IN THE RANGE ',I3,' TO ',I3,$)
                  IF (HIST(K/10) .EQ. 0) GO TO 150
                  WRITE(5,145) (STARS(M),M=1,HIST(K/10))
145               FORMAT (1H+,2X,80A1)
150       CONTINUE
          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:

  0 IN THE RANGE   0 TO  10

  2 IN THE RANGE  10 TO  20  **

  1 IN THE RANGE  20 TO  30  *

 10 IN THE RANGE  30 TO  40  **********

 13 IN THE RANGE  40 TO  50  *************

 11 IN THE RANGE  50 TO  60  ***********

 19 IN THE RANGE  60 TO  70  *******************

 35 IN THE RANGE  70 TO  80  ***********************************

 46 IN THE RANGE  80 TO  90  **********************************************

 17 IN THE RANGE  90 TO 100  *****************

THE TOTAL NUMBER OF TEST SCORES = 155
```

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
        TYPE 50
50      FORMAT (1X,'THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF',
        1  ' AN INTEGER < 32768.'/' ENTERING A NEGATIVE OR ZERO',
        2  ' NUMBER TERMINATES EXECUTION.'/)
80      TYPE 100
100     FORMAT (/' ENTER #   ',$)
        ACCEPT 105,NUMBER
105     FORMAT (I5)
        IF (NUMBER .LE. 0) STOP
        ISQRT = SQRT(FLOAT(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     TYPE 250,NUMBER
250     FORMAT (1X,I5,' IS A PRIME NUMBER'/)
        GO TO 80
300     IF (HOLD .GT. 1) TYPE 350,HOLD
350     FORMAT (1H ,I5)
        GO TO 80
400     IFLAG = 1
        HOLD = HOLD/P
        IF (HOLD .EQ. 1) GO TO 500
        TYPE 450,P
450     FORMAT (1H ,I5,'*')
        GO TO 205
500     TYPE 350,P
        GO TO 80
        END
```

```
        FUNCTION NPRIME(OLD)
        DIMENSION MPRIME(46)
        INTEGER OLD
        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,
        2  127,131,137,139,149,151,157,163,167,173,179,181,
        3  191,193,197,199/
        IF (OLD .EQ. 1) N = 0
        N = N + 1
        NPRIME = MPRIME(N)
        RETURN
        END
```

APPENDIX C

EXAMPLE 3 SAMPLE RUN:

```
THIS IS A PROGRAM TO FIND THE PRIME FACTORS OF AN INTEGER < 32768.
ENTERING A NEGATIVE OR ZERO NUMBER TERMINATES EXECUTION.


ENTER #  16384
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2*
     2

ENTER #  3762
     2*
     3*
     3*
    11*
    19

ENTER #  433
  433 IS A PRIME NUMBER


ENTER #  7263
     3*
     3*
     3*
   269

ENTER #  32767
     7*
    31*
   151

ENTER #  0
```

INDEX

ACCEPT statement, 5-10
ACCESS keyword, 5-23
Actual arguments, 8-1
Adjustable array, 8-8
Adjustable dimensions, 7-5
A field descriptor, 6-9
Alphanumeric literals, 2-10, 6-11
American National Standard FORTRAN, enhancements to, 1-1
Apostrophe character within an alphanumeric literal, 6-11
Auxiliary input/output statements, 5-18
Arguments, 8-1
Arguments, actual, 8-1
Arguments, dummy, 8-1
Arguments in a CALL statement, 4-11
Arithmetic assignment statement, 3-1
Arithmetic expression, data type of an, 2-20
Arithmetic expressions, 2-17
Arithmetic IF statement, 4-4
Arithmetic operators, 2-18
Arithmetic operators, precedence of, 2-19
Arithmetic statement function (ASF), 8-1
Array, adjustable, 8-8
Array, data types of an, 2-16
Array declarator interaction, ENTRY and, 8-8
Array declarators, 2-14
Array declarators, COMMON statements with, 7-8
Array dimensions, 2-13
Array storage, 2-15
Array elements, 2-13
Array name, 2-13
Array names, unsubscripted, 2-16 2-17
Arrays, 2-13
Arrays, EQUIVALENCE and LOGICAL*1, 7-11
ASCII character code, A-2
ASF reference, 8-2
Assigned GO TO statement, 4-3
Assigning Hollerith data to variables, 2-12
ASSIGN statement, 3-5
Assignment statement, logical, 3-4
Assignment statements, 3-1
Assignment statements, conversion rules for, 3-3
ASSOCIATEVARIABLE keyword, 5-26

BACKSPACE statement, 5-18
Basic external functions, 8-15
Basic real constant, 2-6
Binary operators, 2-18
Blank common block, 7-6
BLOCK DATA statement, 8-10
BLOCK DATA subprogram, 8-10
BUFFERCOUNT keyword, 5-24

CALL statement, 4-11, 7-5
CALL statement, arguments in a, 4-11
Carriage control, 6-17
Carriage control characters, 6-17
CARRIAGECONTROL keyword, 5-26
Categories of READ and WRITE statements, 5-1
Character set, FORTRAN, 1-4, A-1
CLOSE statement, 5-27
Coding forms, FORTRAN, 1-5
Comment indicator, 1-7
Comment lines, 1-4
Common block, blank, 7-6
Common block, named, 7-6
COMMON statement, 7-6
COMMON statements with array declarators, 7-8
Complex, 2-3
Complex constants, 2-8
Complex I/O, 6-13
Computed GO TO statement, 4-2
Constant, basic real, 2-6
Constants, 2-5
  complex, 2-8
  double precision, 2-7
  Hollerith, 2-9
  logical, 2-9
  octal, 2-8
Continuation field, 1-8
Continuation indicator, 1-8
Continuation line, 1-3
CONTINUE statement, 4-11
Control statements, 4-1
Control transfers in DO loops, 4-9
Control variable, 4-6
Conversion rules for assignment statements, 3-3

DATA statement, 7-13
Data type by definition, 2-11
Data type by implication, 2-12
Data type of an arithmetic expression, 2-20
Data type of an array, 2-16

Data type storage requirements,
2-4
Data types, 2-2
Debug statement indicator, 1-7
DEFINE FILE statement, 5-14
D field descriptor, 6-6
Difference between functions and
subroutines, 8-1
Dimension declarators, 2-14
expressions as, 7-6
upper and lower bound, 2-14
Dimensions, adjustable, 7-5
DIMENSION statement, 7-4, 7-5
DISPOSE keyword, 5-26
DO lists, implied, 5-4
$ descriptor, 6-13
DO loop,
extended range of, 4-9
termination of a, 4-7
DO loops,
control transfers in, 4-9
nested, 4-8
DO statement, 4-6
Double precision, 2-3
Double precision constants, 2-7
Dummy arguments, 2-2, 8-1

E field descriptor, 6-5
Elements of a FORTRAN program, 1-3
ENCODE and DECODE statements, 5-27
ENDFILE statement, 5-19
END=S specification, 5-17
END statement, 4-13
Enhancements to American National
Standard FORTRAN, 1-1
ENTRY and array declarator inter-
action, 7-11
ENTRY in function subprograms, 8-8
Entry names, 8-8
ENTRY statement, 8-7
EQUIVALENCE and COMMON interaction,
7-11
EQUIVALENCE and LOGICAL*1
arrays, 7-11
EQUIVALENCE statement, 7-8
Equivalent, making arrays, 7-9
ERR keyword, 5-24
Evaluation of parenthesized
arithmetic expressions, 2-23
Examples, FORTRAN programming, C-1
Executable statements, 1-3
EXIT, 4-12
Exponentiation, 2-18
Expression operators, B-1
Expressions, 2-17
logical, 2-23
relational, 2-22
Expressions as dimension declara-
tors, 7-6
Extended range of a DO loop, 4-9
EXTENDSIZE keyword, 5-25
External field separators, 6-18
EXTERNAL statement, 7-12

Field descriptor, 6-1
A, 6-9
D, 6-6
E, 6-5
F, 6-4
G, 6-7
H, 6-10
I, 6-2
L, 6-8
O, 6-3
Q, 6-12
T, 6-12
X, 6-11
Field descriptors, 6-2
Field separator, 6-1
Field separators, external, 6-18
FIND statement, 5-19
Format control, 6-20
Format specification, 6-1
Format specification separators,
6-18
Format specifiers, 5-3
FORMAT statement, 6-1
Format statements, summary of rules
for, 6-21
Formatted direct access input/
output, 5-15
Formatted directed access READ
statement, 5-15
Formatted direct access WRITE
statement, 5-16
Formatted sequential input/output,
5-8
Formatted sequential READ state-
ment, 5-8
Formatted sequential WRITE state-
ment, 5-9
Formatting a FORTRAN line, 1-5
Formatting a line with a TAB
character, 1-6
FORM keyword, 5-23
FORTRAN character set, 1-4, A-1
FORTRAN, enhancements to American
National Standard, 1-1
FORTRAN language summary, B-1
FORTRAN library functions, 8-11
FORTRAN library function (table),
8-12
FORTRAN programming examples, C-1
FORTRAN statement components, 2-1
Function references, processor-
defined, 8-15
Functions and subroutines,
difference between, 8-1
Functions,
basic external, 8-15
intrinsic, 8-15
Functions (table),
FORTRAN library, B-15
generic and processor-defined,
B-17
FUNCTION statement, 8-4
FUNCTION subprogram, 8-3
Function subprograms, ENTRY in,
8-8

Generic and processor-defined
    functions (table), B-17
Generic and processor-defined
    function usage, 8-16
Generic function names, 8-14
Generic function name summary, 8-15
Generic function references, 8-14
G field descriptor, 6-7
GO TO statement,
  assigned, 4-3
  computed, 4-2
  unconditional, 4-2
GO TO statements, 4-1
Grouping and group repeat
    specifications, 6-15
Group repeat count, 6-15

H field descriptor, 6-10
Hollerith constants, 2-9

I field descriptor, 6-2
IF statement,
  arithmetic, 4-4
  logical, 4-5
IF statements, 4-4
IMPLICIT statement, 2-12, 7-1
Implied DO lists, 5-4
Increment parameter, 4-6, 4-7
Indicator,
  comment, 1-7
  continuation, 1-8
Initial parameter, 4-7
INITIALSIZE keyword, 5-25
Input/output,
  devices, 5-2
  lists, 5-3
  records, 5-3
  statements, 5-1
Integer, 2-3
Integer constant, negative, 2-5
Integer constants, 2-5
Intrinsic functions, 8-15
Iteration count, 4-7

Keyword,
  ACCESS, 5-23
  ASSOCIATEVARIABLE, 5-26
  BUFFERCOUNT, 5-24
  CARRIAGECONTROL, 5-26
  DISPOSE, 5-26
  ERR, 5-24
  EXTENDSIZE, 5-25
  FORM, 5-23
  INITIALSIZE, 5-25
  MAXREC, 5-26
  NAME, 5-22
  NOSPANBLOCKS, 5-25
  READONLY, 5-23
  RECORDSIZE, 5-24
  SHARED, 5-25
  TYPE, 5-22
  UNIT, 5-22
Keywords in the OPEN statement, 5-21

Label, statement, 1-4, 1-7
L field descriptor, 6-8
Library functions, B-17
  FORTRAN, 8-11
Library functions (table), FORTRAN,
    8-12
Line, continuation, 1-3
Line (definition), 1-3
Literals, alphanumeric, 2-10
Logical, 2-3
Logical assignment statement, 3-4
Logical constants, 2-9
Logical expressions, 2-23
Logical IF statement, 4-5
Logical operators, 2-23
Logical unit number, 5-2
LOGICAL*1 variables, 2-13

Making arrays equivalent, 7-9
MAXREC keyword, 5-26

Named common block, 7-6
NAME keyword, 5-22
Names,
  entry, 8-8
  generic function, 8-14
Negative integer constant, 2-5
Nested DO loops, 4-8
NOSPANBLOCKS keyword, 5-25
Number, statement, 1-7

Object time format, 6-19
Octal constants, 2-8
O field descriptor, 6-3
OPEN statement, 5-20
OPEN statement, keywords in the,
    5-21
Operators,
  arithmetic, 2-18
  binary, 2-18

PARAMETER statement, 7-14
Parentheses, use of, 2-19
Parenthesized arithmetic expressions,
    evaluation of, 2-23
PAUSE statement, 4-12
PDF (processor defined functions),
    8-15
Precedence of arithmetic operators,
    2-19

PRINT statement, 5-11
Processor-defined function
    references, 8-15
Processor-defined functions
    (table), generic and, 8-17
Processor-defined function usage,
    generic and, 8-16
PROGRAM statement, 7-15
Program unit structure, 1-9

Q field descriptor, 6-12

RADIX-50 character code, A-3
Radix-50 constants, 2-10
READ and WRITE statements,
    categories of, 5-1
READONLY keyword, 5-23
READ statement,
  formatted direct access, 5-15
  formatted sequential, 5-8
  unformatted direct access, 5-12
  unformatted sequential, 5-6
Real, 2-3
Real constant, basic, 2-6
Records, input/output, 5-3
RECORDSIZE keyword, 5-24
Relational expressions, 2-22
Relational operator, 2-22
Required order of statements and
    lines, 1-9
RETURN statement, 4-12
REWIND statement, 5-18

Scale factor, 6-14
Separators, format specification,
    6-18
Sequence number field, 1-8
SHARED keyword, 5-25
Short field termination, 6-19
Simple I/O lists, 5-3
Space character in a FORTRAN
    statement, 1-7
Specification statements, 7-1
Statement,
  ACCEPT, 5-10
  arithmetic assignment, 3-1
  arithmetic IF, 4-4
  ASSIGN, 3-5
  assigned GO TO, 4-3
  BACKSPACE, 5-18
  BLOCK DATA, 8-10
  CALL, 4-11, 7-5
  CLOSE, 5-27
  COMMON, 7-6
  computed GO TO, 4-2
  CONTINUE, 4-11
  DATA, 7-13
  DEFINE FILE, 5-14
  DIMENSION, 7-4, 7-5
  DO, 4-6
  END, 4-13
  ENDFILE, 5-19
  ENTRY, 8-7
  EQUIVALENCE, 7-8
  EXTERNAL, 7-12
  FIND, 5-19
  FORMAT, 6-1
  formatted direct access READ, 5-15
  formatted sequential READ, 5-8
  formatted sequential WRITE, 5-9
  FUNCTION, 8-4
  IMPLICIT, 7-1
  logical assignment, 3-4
  logical IF, 4-5
  OPEN, 5-20
  PARAMETER, 7-14
  PAUSE, 4-12
  PRINT, 5-11
  PROGRAM, 7-15
  RETURN, 4-12
  REWIND, 5-18
  STOP, 4-13
  SUBROUTINE, 8-5
  TYPE, 5-11
  unconditional GO TO, 4-2
  unformatted direct access READ,
    5-12
  unformatted direct access WRITE,
    5-13
  unformatted sequential READ, 5-6
  unformatted sequential WRITE, 5-6
Statement components, FORTRAN, 2-1
Statement field, 1-8
Statement label, 1-4, 1-7
Statement number, 1-7
Statements, 1-3
  executable, 1-3
  summary of, B-2
  type declaration, 7-2
Statements and lines, required
    order of, 1-9
STOP statement, 4-13
Storage requirements, data type,
    2-4
Storage unit, 2-3
Subprogram,
  BLOCK DATA, 8-10
  FUNCTION, 8-3
Subprograms, 8-1
Subroutines, difference between
    functions and, 8-1
SUBROUTINE statement, 8-5
SUBROUTINE subprogram, 8-5
Subscripts, 2-13, 2-16
Summary of rules for format state-
    ments, 6-21
Summary of statements, B-2
Symbolic names, 2-2

TAB character, formatting a line
    with a, 1-6
Terminal statement, 4-7
Termination of a DO loop, 4-7
Text editor, using a, 1-6

T field descriptor, 6-12
Transfer of control, 4-1
Transfer of control on end-of-file
    or error conditions, 5-16
Type declaration statements, 7-2
TYPE keyword, 5-22
TYPE statement, 5-11

Unconditional GO TO statement,
    4-2
Unformatted direct access
    input/output, 5-12
Unformatted direct access READ
    statement, 5-12
Unformatted direct access WRITE
    statement, 5-13
Unformatted sequential input/output,
    5-6
Unformatted sequential READ
    statement, 5-6
Unformatted sequential WRITE
    statement, 5-6
UNIT keyword, 5-22
Unsubscripted array name in an
    I/O list, 5-4
Unsubscripted array names, 2-16,
    2-17
Upper and lower bound dimension
    declarators, 2-14
Use of parentheses, 2-19
User-written subprograms, 8-1
Using FORTRAN coding forms, 1-5

Variable format expressions, 6-16
Variable names, 2-11
Variables, 2-11
  assigning Hollerith data to, 2-12
  LOGICAL*1, 2-13

WRITE statement,
  formatted direct access, 5-16
  formatted sequential, 5-9
  unformatted direct access, 5-13
  unformatted sequential, 5-6

X field descriptor, 6-11

HOW TO OBTAIN SOFTWARE INFORMATION

SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes newsletters and Software Performance Summaries (SPS) for the various Digital products. Newsletters are published monthly, and contain announcements of new and revised software, programming notes, software problems and solutions, and documentation corrections. Software Performance Summaries are a collection of existing problems and solutions for a given software system, and are published periodically. For information on the distribution of these documents and how to get on the software newsletter mailing list, write to:

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

SOFTWARE PROBLEMS

Questions or problems relating to Digital's software should be reported to a Software Support Specialist. A specialist is located in each Digital Sales Office in the United States. In Europe, software problem reporting centers are in the following cities.

Reading, England	Milan, Italy
Paris, France	Solna, Sweden
The Hague, Holland	Geneva, Switzerland
Tel Aviv, Israel	Munich, West Germany

Software Problem Report (SPR) forms are available from the specialists or from the Software Distribution Centers cited below.

PROGRAMS AND MANUALS

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center.

Digital Equipment Corporation
Software Distribution Center
146 Main Street
Maynard, Massachusetts 01754

Digital Equipment Corporation
Software Distribution Center
1400 Terra Bella
Mountain View, California 94043

Outside of the United States, orders should be directed to the nearest Digital Field Sales Office or representative.

USERS SOCIETY

DECUS, Digital Equipment Computer Users Society, maintains a user exchange center for user-written programs and technical application information. A catalog of existing programs is available. The society publishes a periodical, DECUSCOPE, and holds technical seminars in the United States, Canada, Europe, and Australia. For information on the society and membership application forms, write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

DECUS EUROPE
Digital Equipment Corp. International (Europe)
P.O. Box 340
1211 Geneva 26
Switzerland

PDP-11 FORTRAN
Language Reference Manual
DEC-11-LFLRA-B-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

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 do not require a written reply, please check here. ☐

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

DIGITAL EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS 01754