decsystem10

FORTRAN-10 LANGUAGE MANUAL

Second Edition

digital

decsystem10

FORTRAN-10 language manual

This manual reflects the software as of Version 1A of the FORTRAN-10 Compiler and Version 2 of the FORTRAN-10 Object Time System (FOROTS).

Additional copies of this manual may be ordered from: Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754 Order Code: DEC-10-LFORA-B-D

digital equipment corporation · maynard, massachusetts

First Edition June 1973
Second Edition January 1974

CONTENTS

		Page
CHAPTER 1	INTRODUCTION	
1.1	INTRODUCTION	1-1
CHAPTER 2	CHARACTERS AND LINES	
2.1	CHARACTER SET	2-1
2.2	STATEMENT, DEFINITION, AND FORMAT	2-3
2.2.1	Statement Label Field and Statement Numbers	2-3
2.2.2	Line Continuation Field	2-3
2.2.3	Statement Field	2-3
2.2.4	Remarks	2-3
2.3	LINE TYPES	2-4
2.3.1	Initial and Continuation Line Types	2-4
2.3.2	Multi-Statement Lines	2-5
2.3.3	Comment Lines and Remarks	2-5
2.3.4	Debug Lines	2-6
2.3.5	Blank Lines	2-6
2.3.6	Line-Sequenced Input	2-6
CHAPTER 3	DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS	
3.1	DATA TYPES	3-1
3.2	CONSTANTS	3-1
3.2.1	Integer Constants	3-2
3.2.2	Real Constants	3-2
3.2.3	Double Precision Constants	3-3
3.2.4	Complex Constants	3-4
3.2.5	Octal Constants	3-4
3.2.6	Logical Constants	3-5
3.2.7	Literal Constants	3-5
3.2.8	Statement Labels	3-5
3.3	SYMBOLIC NAMES	3-5
3.4	VARIABLES	3-6
3.5	ARRAYS	3-7
3.5.1	Array Element Subscripts	3-7
3.5.2	Dimensioning Arrays	3-8
3.5.3	Order of Stored Array Elements	3-9
CHAPTER 4	EXPRESSIONS	
4.1	ARITHMETIC EXPRESSIONS	4-1
4.1.1	Rules for Writing Arithmetic Expressions	4-2
4.2	LOGICAL EXPRESSIONS	4-2
4.2.1	Relational Expressions	4-6
4.3	EVALUATION OF EXPRESSIONS	4-8
4.3.1	Parenthesized Subexpressions	4-8
4.3.2	Hierarchy of Operators	4-8
4.3.3	Mixed Mode Expressions	4-9
4.3.4	Use of Logical Operands in Mixed Mode Expressions	4-10

		Page
CHAPTER 5	COMPILATION CONTROL STATEMENTS	
5.1	INTRODUCTION	5-1
5.2	END STATEMENT	5-1
CHAPTER 6	SPECIFICATION STATEMENT	
6.1	INTRODUCTION	6-1
6.2	DIMENSION STATEMENT	6-1
6.2.1	Adjustable Dimensions	6-2
6.3	TYPE SPECIFICATION STATEMENTS	6-3
6.4	IMPLICIT STATEMENTS	6-4
6.5	COMMON STATEMENT	6-5
6.5.1	Dimensioning Arrays in COMMON Statements	6-6
6.6	EQUIVALENCE STATEMENT	6-6
6.7	EXTERNAL STATEMENT	6-7
CHAPTER 7	DATA STATEMENT	
7.1	INTRODUCTION	7-1
CHAPTER 8	ASSIGNMENT STATEMENTS	
8.1	INTRODUCTION	8-1
8.2	ARITHMETIC ASSIGNMENT STATEMENT	8-1
8.3	LOGICAL ASSIGNMENT STATEMENTS	8-3
8.4	ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT	8-3
CHAPTER 9	CONTROL STATEMENTS	
9.1	INTRODUCTION	9-1
9.2	GO TO CONTROL STATEMENTS	9-1
9.2.1	Unconditional GO TO Statements	9-2
9.2.2	Computed GO TO Statements	9-2
9.2.3	Assigned GO TO Statements	9-2
9.3	IF STATEMENTS	9-3
9.3.1	Arithmetic IF Statements	9-3
9.3.2	Logical IF Statements	9-4
9.3.3	Logical Two-Branch IF Statements	9-4
9.4	DO STATEMENT	9-5
9.4.1	Nested DO Statements	9-6
9.4.2	Extended Range	9-7
9.4.3	Permitted Transfer Operations	9-8
9.5	CONTINUE STATEMENT	9-9
9.6	STOP STATEMENT	9-9
9.7	PAUSE STATEMENT	9-10
9.7.1	T (TRACE) Option	9-10

		Page
CHAPTER 10	I/O STATEMENTS	
10.1	DATA TRANSFER OPERATIONS	10-1
10.2	TRANSFER MODES	10-1
10.2.1	Sequential Mode	10-1
10.2.2	Random Access Mode	10-1
10.2.3	Append Mode	10-2
10.3	I/O STATEMENTS, BASIC FORMATS AND COMPONENTS	10-2
10.3.1	I/O Statement Keywords	10-3
10.3.2	FORTRAN-10 Logical Unit Numbers	10-3
10.3.3	FORMAT Statement References	10-3
10.3.4	I/O List	10-5
10.3.5	The Specification of Records for Random Access	10-6
10.3.6	List-Directed I/O	10-6
10.3.7	NAMELIST I/O Lists	10-8
10.4	OPTIONAL READ/WRITE ERROR EXIT AND END-OF-FILE ARGUMENTS	10-8
10.5	READ STATEMENTS	10-9
10.5.1	Sequential Formatted READ Transfers	10-9
10.5.2	Sequential Unformatted Binary READ Transfers	10-10
10.5.3	Sequential List-Directed READ Transfers	10-10
10.5.4	Sequential NAMELIST-Controlled READ Transfers	10-11
10.5.5	Random Access Formatted READ Transfers	10-11
10.5.6	Random Access Unformatted READ Transfers	10-11
10.6	SUMMARY OF READ STATEMENTS	10-11
10.7	REREAD STATEMENT	10-12
10.8	WRITE STATEMENTS	10-13
10.8.1	Sequential Formatted WRITE Transfers	10-13
10.8.2	Sequential Unformatted WRITE Transfer	10-14
10.8.3	Sequential List-Directed WRITE Transfers	10-14
10.8.4	Sequential NAMELIST-Controlled WRITE Transfers	10-14
10.8.5	Random Access Formatted WRITE Transfers	10-14
10.8.6	Random Access Unformatted WRITE Transfers	10-15
10.9	SUMMARY OF WRITE STATEMENTS	10-15
10.10	ACCEPT STATEMENT	10-15
10.10.1	Formatted ACCEPT Transfers	10-15
10.10.2	ACCEPT Transfers Into FORMAT Statement	10-16
10.11	PRINT STATEMENT	10-16
10.12	PUNCH STATEMENT	10-17
10.13	TYPE STATEMENT	10-18
10.14	FIND STATEMENT	10-18
10.15	ENCODE AND DECODE STATEMENTS	10-19
10.15.1	ENCODE Statement	10-19
10.15.2	DECODE Statement	10-20
10.15.3	Example of ENCODE/DECODE Operations	10-20
10.16	SUMMARY OF I/O STATEMENTS	10-21

CONTENTS (Cont)

Page

CHAPTER 11 NAMELIST STATEMENTS

11.1 INTRODUCTION 11-1
11.2 NAMELIST STATEMENT 11-1
11.2.1 NAMELIST-Controlled Input Transfers 11-2
11.2.2 NAMELIST-Controlled Output Transfers 11-3

CHAPTER 12 FILE CONTROL STATEMENTS

12.1 INTRODUCTION 12-1
12.2 OPEN AND CLOSE STATEMENTS 12-1
12.2.1 Options for OPEN and CLOSE Statements 12-2
12.2.2 Summary of OPEN/CLOSE Statement Options 12-9

CHAPTER 13 FORMAT STATEMENT

13.1 INTRODUCTION 13-1
13.1.1 FORMAT Statement, General Form 13-1
13.2 FIELD DESCRIPTORS 13-2
13.2.1 Numeric Field Descriptors 13-4
13.2.2 Interaction of Field Descriptors With I/O List Variables During Transfer 13-6
13.2.3 G, General Numeric Conversion Code 13-7
13.2.4 Numeric Fields with Scale Factors 13-7
13.2.5 Logical Field Descriptors 13-9
13.2.6 Variable Numeric Field Widths 13-9
13.2.7 Alphanumeric Field Descriptors 13-9
13.2.8 Transferring Alphanumeric Data Directly Into or From FORMAT Statements . . . 13-10
13.2.9 Mixed Numeric and Alphanumeric Fields 13-11
13.2.10 Multiple Record Specifications 13-12
13.2.11 Record Formatting Field Descriptors 13-13
13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS 13-14

CHAPTER 14 DEVICE CONTROL STATEMENTS

14.1 INTRODUCTION 14-1
14.2 REWIND STATEMENT 14-2
14.3 UNLOAD STATEMENT 14-2
14.4 BACKSPACE STATEMENT 14-2
14.5 END FILE STATEMENT 14-2
14.6 SKIP RECORD STATEMENT 14-3
14.7 SKIP FILE STATEMENT 14-3
14.8 BACKFILE STATEMENT 14-3
14.9 SUMMARY OF DEVICE CONTROL STATEMENTS 14-3

CHAPTER 15 SUBPROGRAM STATEMENTS

15.1 INTRODUCTION 15-1
15.1.1 Dummy and Actual Arguments 15-1
15.2 STATEMENT FUNCTIONS 15-3
15.3 INTRINSIC FUNCTIONS 15-3

		Page
15.4	EXTERNAL FUNCTIONS	15-5
15.4.1	Basic External Functions	15-6
15.4.2	Generic Function Names	15-6
15.5	SUBROUTINE SUBPROGRAMS	15-7
15.5.1	Referencing Subroutines (CALL Statement)	15-9
15.5.2	FORTRAN-10 Supplied Subroutines	15-10
15.6	RETURN STATEMENT AND MULTIPLE RETURNS	15-10
15.6.1	Referencing External FUNCTION Subprograms	15-12
15.7	MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT)	15-13
CHAPTER 16	**BLOCK DATA SUBPROGRAMS**	
16.1	INTRODUCTION	16-1
16.2	BLOCK DATA STATEMENT	16-1
APPENDIX A	**ASCII-1968 CHARACTER CODE SET**	
APPENDIX B	**WRITING USER PROGRAMS**	
B.1	RUNNING THE FORTRAN-10 COMPILER	B-1
B.1.1	Switches Available with the FORTRAN-10 Compiler	B-1
B.1.2	Monitor Commands	B-2
B.2	READING A FORTRAN-10 LISTING	B-3
B.3	ERROR REPORTING	B-8
B.4	WRITING EFFECTIVE FORTRAN-10 PROGRAMS	B-8
B.4.1	General Programming Considerations	B-8
B.4.1.1	Accuracy and Range of Double Precision Numbers	B-8
B.4.1.2	Writing FORTRAN-10 Programs for Execution on Non-DEC Machines	B-8
B.4.2	Storage of Arrays	B-9
B.4.3	Use of COMMON	B-9
B.4.4	Use of EQUIVALENCE Statements	B-10
B.4.5	Use of ENTRY Statements	B-11
B.4.6	Using Floating Point DO Loops	B-12
B.4.7	Computation of DO Loop Iterations	B-12
B.4.8	List-Directed I/O	B-13
B.4.9	Subroutines–Programming Considerations	B-14
B.4.10	Reordering of Computations	B-14
B.5	FORTRAN-10 GLOBAL OPTIMIZER	B-15
B.5.1	Optimization Techniques	B-16
B.5.1.1	Elimination of Common Subexpressions	B-16
B.5.1.2	Reduction of Operator Strength	B-17
B.5.1.3	Removal of Constant Computation From Loops	B-17
B.5.1.4	Constant Folding and Propagation	B-18
B.5.1.5	Removal of Inaccessible Code	B-19
B.5.1.6	Global Register Allocation	B-19
B.5.2	Improper Function References	B-19
B.5.3	Programming Techniques for Effective Optimization	B-19

		Page
B.6	INTERFACING WITH NON–FORTRAN–10 PROGRAMS AND FILES	B-19
B.6.1	Calling Sequences	B-19
B.6.2	Accumulator Usage	B-20
B.6.3	Argument Lists	B-21
B.6.4	Argument Types	B-22
B.6.5	Description of Arguments	B-22
B.6.6	Converting Existing MACRO-10 Libraries for use with FORTRAN-10	B-24
B.6.7	Mixing FORTRAN-10 and F40 Compiled Programs	B-29
B.6.8	Interaction with COBOL-10	B-29
B.6.8.1	Calling FORTRAN-10 Subprograms as COBOL-10 Programs	B-29
B.6.8.2	Calling COBOL-10 Subroutines From FORTRAN-10 Programs	B-31
B.6.9	FOROTS/FORSE Compatibility	B-31
B.6.9.1	FORTRAN-10/F40 Data File Compatibility	B-31
B.6.9.2	Conversion of FOROTS-Developed Data Files Into a Form Acceptable to FORSE	B-32
B.6.9.3	General Restrictions	B-34
APPENDIX C	**FOROTS**	
C.1	INTRODUCTION	C-1
C.1.1	Hardware Requirements	C-1
C.1.2	Software Requirements	C-1
C.2	FEATURES OF FOROTS	C-1
C.3	ERROR PROCESSING	C-3
C.4	INPUT/OUTPUT FACILITIES	C-3
C.4.1	Input/Output Channels	C-3
C.4.2	File Access Modes	C-3
C.4.2.1	Sequential Transfer Mode	C-3
C.4.2.2	Random Access Mode	C-4
C.5	ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS	C-4
C.5.1	ASCII Data Files	C-4
C.5.2	ASCII Data Files with Line Sequence Numbers	C-4
C.5.3	FORTRAN Binary Data Files	C-5
C.5.4	Mixed Mode Data Files	C-5
C.5.5	Image Binary Files	C-5
C.6	USING FOROTS AS A GENERAL I/O SYSTEM	C-6
C.6.1	FOROTS Entry Points	C-6
C.6.2	Calling Sequences	C-6
C.6.3	MACRO Calls for FOROTS Functions	C-7
C.6.3.1	Initialization of FOROTS	C-8
C.6.3.2	ENCODE/DECODE Calling Sequences	C-8
C.6.3.3	Formatted/Unformatted Transfer Statements, Sequential Access Calling Sequences	C-9
C.6.3.4	NAMELIST Data Transfer Statements, Sequential Access Calling Sequences	C-10
C.6.3.5	Formatted/Unformatted Data Transfer Statements, Random Access Calling Sequences	C-11

		Page
C.6.3.6	Calling Sequences for Statements Which Use Default Devices	C-12
C.6.3.7	Calling Sequences for Statements Which Position Magnetic Tape Units	C-13
C.6.3.8	List Directed Input/Output Statements	C-14
C.6.3.9	Input/Output Data Lists	C-14
C.6.3.10	OPEN and CLOSE Statements, Calling Sequences	C-16
C.6.3.11	Memory Allocation Routines	C-17
C.6.3.12	Software Channel Allocation and Deallocation Routines	C-18
C.7	DETAILED DESCRIPTION	C-19
C.7.1	FOROTS Source Files	C-20
C.7.1.1	FORPRM Parameter File	C-20
C.7.1.2	FORINI Initialization File	C-20
C.7.1.3	FORCNV Data Conversion File	C-20
C.7.1.4	FORTRP Trap Handler	C-21
C.7.1.5	FORERR Error Routine	C-21
C.7.1.6	FOROTS Main I/O Processing and Control File	C-21
C.8	FOROTS CORE REQUIREMENTS	C-23
C.8.1	Core and Data File Protection	C-24
C.9	LOGICAL/PHYSICAL DEVICE ASSIGNMENTS	C-24
APPENDIX D	**DEBUGGING FORTRAN PROGRAMS**	
D.1	LOADING AND STARTING FORDDT	D-1
D.2	FORDDT COMMANDS	D-1
D.2.1	Starting the Program	D-2
D.2.2	Stopping the Program	D-2
D.2.3	Opening Subprograms	D-2
D.2.4	Changing the Values of Variables	D-3
D.2.5	Grouping Parameters for Commands	D-4
D.2.6	Specifying Typeout Modes	D-4
D.2.7	Displaying Values	D-5
D.2.8	Setting Pauses (Breakpoints)	D-5
D.2.9	Removing Pauses (Breakpoints)	D-6
D.2.10	Continuing After a Pause (Breakpoint)	D-7
D.2.11	Obtaining Information	D-7
D.2.12	Tracing Subroutine Calls	D-10
D.2.13	Entering and Leaving DDT	D-11

TABLES

Table No.	Title	Page
1-1	FORTRAN-10 Statement Categories	1-2
2-1	FORTRAN-10 Character Set	2-1
3-1	Constants	3-2
3-2	Use of Symbolic Names	3-6
4-1	Arithmetic Operations and Operators	4-1
4-2	Type of the Resultant Obtained From Mixed Mode Operations	4-3
4-3	Permitted Base/Exponent Type Combinations	4-4
4-4	Logical Operators	4-4
4-5	Logical Operations, Truth Table	4-5
4-6	Binary Logical Operations, Truth Table	4-6
4-7	Relational Operators and Operations	4-7
4-8	Hierarchy of FORTRAN-10 Operators	4-9
8-1	Rules for Conversion in Mixed Mode Assignments	8-2
10-1	FORTRAN-10 Logical Device Assignments	10-4
10-2	Summary of Read Statements	10-12
10-3	Summary of WRITE Statements	10-15
10-4	Summary of FORTRAN-10 I/O Statements	10-22
12-1	OPEN/CLOSE Statement Arguments	12-9
13-1	FORTRAN-10 Conversion Codes	13-3
13-2	Action of Field Descriptors on Sample Data	13-5
13-3	Numeric Field Codes	13-6
13-4	Descriptor Conversion of Real and Double Precision Data According to Magnitude	13-8
13-5	FORTRAN-10 Print Control Characters	13-14
14-1	Summary of FORTRAN-10 Device Control Statements	14-3
15-1	Intrinsic Functions	15-4
15-2	Basic External Functions	15-8
15-3	FORTRAN-10 Library Subroutines	15-15
B-1	FORTRAN-10 Compiler Switches	B-2
B-2	Argument Types and Type Codes	B-22
B-3	Upward Compatibility (FORSE TO FOROTS)	B-32
B-4	Downward Compatibility (FOROTS TO FORSE)	B-33
C-1	FORTRAN Device Table	C-24
D-1	Modes for ACCEPT Values	D-3
D-2	Typeout Modes	D-4

PREFACE

The FORTRAN language set as implemented by the new DECsystem-10 FORTRAN-10 Language Processing System (referred to as FORTRAN-10) is described in this manual.

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

The descriptions of the FORTRAN-10 extensions and additions to the standard FORTRAN language set are printed in boldface italic type.

Operating procedures and descriptions of the DECsystem-10 programming environment for FORTRAN programmers are given in Appendix B. The FORTRAN-10 Object Time System (FOROTS) is described in Appendix C.

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The FORTRAN-10 language set is compatible with and encompasses the standard set described in "American National Standard FORTRAN, X3.9-1966" (referred to as the 1966 ANSI standard set). FORTRAN-10 also provides many extensions and additions to the standard set which greatly enhance the usefulness of FORTRAN-10 and increases its compatibility with FORTRAN language sets implemented by other major computer manufacturers. In this manual the FORTRAN-10 extensions and additions to the 1966 ANSI standard set are printed in ***boldface italic type.***

A FORTRAN-10 source program consists of a set of statements constructed using the language elements and the syntax described in this manual. A given FORTRAN-10 statement will perform any one of the following functions:

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

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

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

FORTRAN-10 statements are comprised of key words (i.e., words which are recognized by the compiler) used with elements of the language set: constants, variables, and expressions. There are two basic types of FORTRAN-10 statements: executable and nonexecutable.

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

In this manual the FORTRAN-10 statements are grouped into twelve categories, each of which is described in a separate chapter. The name, definition, and chapter reference for each statement category are given in Table 1-1.

The basic FORTRAN-10 language elements (i.e., constants, variables, and expressions), the character set from which they may be formed, and the rules which govern their construction and use are described in Chapters 2 through 4.

Table 1-1
FORTRAN-10 Statement Categories

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

CHAPTER 2
CHARACTERS AND LINES

2.1 CHARACTER SET

The digits, letters, and symbols recognized by FORTRAN-10 are listed in Table 2-1. The remainder of the ASCII-1968 character set[1], although accepted by FORTRAN-10, is regarded as meaningless, and if used in a statement will cause an information-only (nonfatal) message to be printed or displayed at the user's terminal during program compilation. Null characters are ignored completely.

NOTE

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

Table 2-1
FORTRAN-10 Character Set

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

(continued)

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

Table 2-1 (Cont)
FORTRAN-10 Character Set

Digits	
0	5
1	6
2	7
3	8
4	9

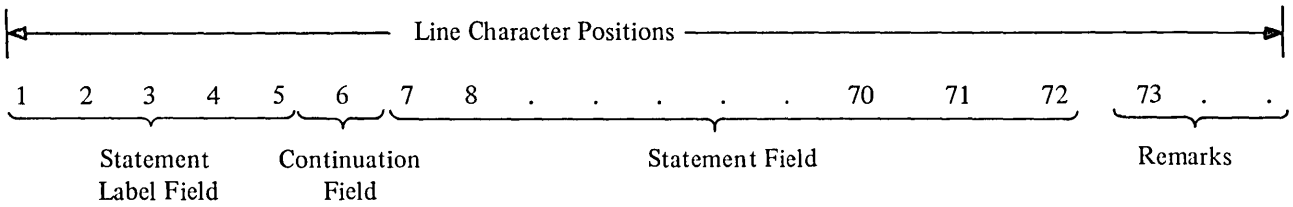
Symbols	
! Exclamation Point	/ Slant (slash)
" Quotation Marks	: Colon
# Number Sign	; Semicolon
$ Dollar Sign	< Less Than
% Percent	= Equals
& Ampersand	> Greater Than
' Acute Accent	? Question Mark
(Opening Parenthesis	@ Commercial At Sign
) Closing Parenthesis	[Opening Bracket
* Asterisk	\ Reverse Slant
+ Plus	] Closing Bracket
, Comma	^ Circumflex
- Hyphen (Minus)	— Underline
. Period (Decimal Point)	

Non-Printing Characters
Line Termination Characters
Line Feed
Form Feed
Vertical Tab
Line Formatting Characters
Carriage Return
Horizontal Tab
Blank

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

2.2 STATEMENT, DEFINITION, AND FORMAT

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



2.2.1 Statement Label Field and Statement Numbers

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

When source programs are entered into a DECsystem-10 system via a standard user terminal, an initial tab may be used to skip all or part of the label field.

If an initial tab is encountered during compilation, FORTRAN-10 examines the character immediately following the tab to determine the type of line being entered. If the character following the tab is one of the digits 1 through 9, FORTRAN-10 considers the line as a continuation line and the second character after the tab as the first character of the statement field. If the character following the tab is other than one of the digits 1 through 9, FORTRAN-10 considers the line to be an initial line and the character following the tab is considered to be the first character of the statement field. The character following the initial tab is considered to be in character position 6 in a continuation line, and in character position 7 in an initial line.

2.2.2 Line Continuation Field

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

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

2.2.3 Statement Field

Any FORTRAN-10 statement may appear in this field. Blanks (spaces) and tabs do not affect compilation of the statement and may be used freely in this field for appearance purposes, with the exception of textual data given within either a literal or Hollerith specification where blanks and tabs are significant characters.

2.2.4 Remarks

In lines comprised of 73 or more character positions, only the first 72 characters are interpreted by FORTRAN-10. (Note that tabs generally occupy more than one character position.) All other characters in the line (character positions 73, 74 . . . etc.) are treated as remarks and do not affect compilation.

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

2.3 LINE TYPES

A line in a FORTRAN-10 source program can be

a. an initial line

b. a continuation line

c. *a multi-statement line*

d. a comment line

e. *a debug line*

f. a blank line.

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

2.3.1 Initial and Continuation Line Types

A FORTRAN-10 statement may occupy the statement fields of up to 20 consecutive lines. The first line in a multi-line statement group is referred to as the "initial" line; the succeeding lines are referred to as continuation lines.

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

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

Continuation lines cannot be assigned statement numbers; they are identified by any character (except for a blank or zero) placed in character position 6 of the line (i.e., continuation line field).

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

Following is an example of a four line FORTRAN-10 FORMAT statement using initial tabs:

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

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

2.3.2 Multi-Statement Lines

More than one FORTRAN-10 statement may be written in the statement field of one line. The rules for structuring a multi-statement line are:

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

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

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

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

An example of a multi-statement is:

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

2.3.3 Comment Lines and Remarks

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

The rules for structuring a comment line are:

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

b. The text may be written into character positions 2 through 72 of the line.

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

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

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

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

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

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

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

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

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

2.3.4 Debug Lines

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

2.3.5 Blank Lines

Lines consisting of only blanks, tabs, or no characters may be inserted anywhere in a FORTRAN-10 source program except between an initial and continuation line, or between two continuation lines. Blank lines are used for formatting purposes only; they cause blank lines to appear in their corresponding positions in object program listings; otherwise, they are ignored by the compiler.

2.3.6 Line-Sequenced Input

FORTRAN-10 optionally accepts DECsystem-10 line-sequenced files as produced by LINED or BASIC. These sequence numbers are used in place of the listing line numbers normally generated by FORTRAN-10.

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

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

3.1 DATA TYPES

The data types permitted in FORTRAN-10 source programs are

a. integer

b. real

c. double precision

d. complex

e. octal

f. double octal

g. literal

h. statement label, and

i. logical.

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

3.2 CONSTANTS

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

The constants permitted in FORTRAN-10 are listed in Table 3-1.

Table 3-1
Constants

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

3.2.1 Integer Constants

An integer constant is a string of from one to eleven digits which represents a whole decimal number (i.e., a number without a fractional part). Integer constants must be within the range of -2^{35} to $+2^{35}-1$ (i.e., -34359738368 to +34359738367). Positive integer constants may optionally be signed; negative integer constants must be signed. Decimal points, commas, or other symbols are not permitted in integer constants (except for a preceding sign, + or -). Examples of *valid* integer constants are:

```
 345
+345
-345
```

Examples of *invalid* integer constants are:

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

3.2.2 Real Constants

A real constant may have any of the following forms:

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

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

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

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

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

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

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

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

The following are examples of *invalid* real constants.

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

3.2.3 Double Precision Constants

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

a. Double precision constants depending on their magnitude have precision to either 15 to 17 places (system with a KA10 Processor) or 16 to 18 places (system with a KI10 Processor), rather than the 8-digit precision obtained for real constants.

b. Each double precision constant occupies two storage locations.

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

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

Processor	Range
KA10	0.14×10^{-31} to $1.7 \times 10^{+31}$
KI10	0.14×10^{-38} to $1.7 \times 10^{+38}$

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

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

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

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

3.2.4 Complex Constants

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

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

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

3.2.5 Octal Constants

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

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

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

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

All octal constants must be

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

b. signed if negative but optionally signed if positive.

The following are examples of valid octal constants:

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

The following are examples of invalid octal constants:

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

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

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

3.2.6 Logical Constants

The Boolean values of truth and falsehood are represented in FORTRAN-10 source programs as the logical constants .TRUE. and .FALSE.. Logical constants are always written enclosed by periods as in the preceding sentence.

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

3.2.7 Literal Constants

A literal constant may be either of the following:

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

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

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

NOTE

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

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

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

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

The 7-bit ASCII characters which comprise a literal constant are stored left-justified (starting in the high order word of a 2-word precision or complex literal) with blanks placed in empty character positions.

3.2.8 Statement Labels

Statement labels are numeric identifiers that represent program statement numbers.

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

Statement labels may be used in the argument list of CALL statements and subprogram formal statement lists (Chapter 15).

3.3 SYMBOLIC NAMES

Symbolic names may consist of any alphanumeric combination of from one to six legal FORTRAN-10 characters. *More than six characters may be given but FORTRAN-10 will ignore all but the first six.* The *first* character of a symbolic name *must* be an alphabetic character.

The following are examples of legal symbolic names:

A12345
IAMBIC
ABLE

The following are examples of illegal symbolic names:

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

Symbolic names are used to identify specific items of a FORTRAN-10 source program; these items, together with an example of a symbolic name and text reference for each, are listed in Table 3-2.

Table 3-2
Use of Symbolic Names

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

3.4 VARIABLES

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

If an undefined variable is referenced an unknown value (i.e., garbage) will be obtained.

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

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

a. integer

b. real

c. logical

d. double precision, or

e. complex.

The type of a variable may be declared using either implicit or explicit type declaration statements (Chapter 6). However, if type declaration statements are not used, the following convention is assumed by FORTRAN-10:

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

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

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

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

3.5 ARRAYS

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

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

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

3.5.1 Array Element Subscripts

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

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

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

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

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

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

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

a. IAB (1,5,3)

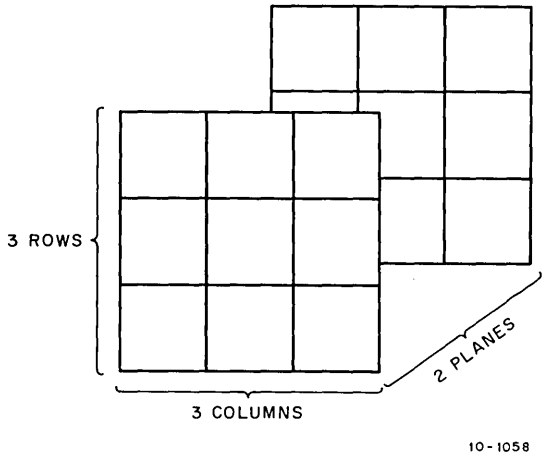
b. ABLE (A)

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

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

3.5.2 Dimensioning Arrays

The size (i.e., number of elements) of an array must be declared in order to enable FORTRAN-10 to reserve the needed amount of locations in which to store the array. Arrays are stored as a series of sequential storage locations. Arrays, however, are visualized and referenced as if they were single or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. For example, the following figure represents a 3-row, 3-column, 2-plane array.

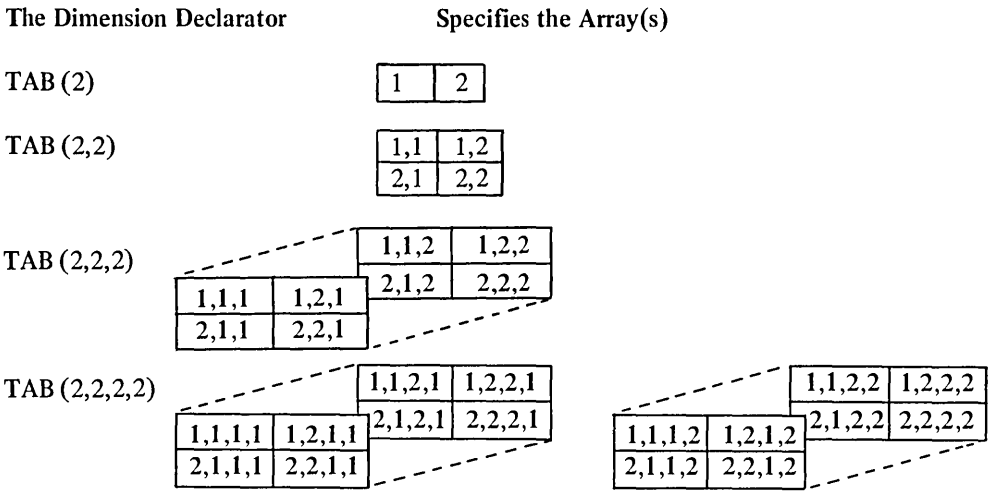


The size (i.e., number of elements) of an array is specified by an array declarator written as a subscripted array name. In an array declarator, however, each subscript quantity is a dimension of the array and must be either an integer, a variable, or an integer constant.

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

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

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



NOTE
FORTRAN-10 permits any number of dimensions in an array declarator.

3.5.3 Order of Stored Array Elements

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

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

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

```
   B(1,1,1)    B(2,1,1)    B(3,1,1) --┐
└-->B(1,2,1)    B(2,2,1)    B(3,2,1) --┐
└-->B(1,3,1)    B(2,3,1)    B(3,3,1) --┐
└-->B(1,1,2)    B(2,1,2)    B(3,1,2) --┐
└-->B(1,2,2)    B(2,2,2)    B(3,2,2) --┐
└-->B(1,3,2)    B(2,3,2)    B(3,3,2) --┐
└-->B(1,1,3)    B(2,1,3)    B(3,1,3) --┐
└-->B(1,2,3)    B(2,2,3)    B(3,2,3) --┐
└-->B(1,3,3)    B(2,3,3)    B(3,3,3)
```

CHAPTER 4
EXPRESSIONS

4.1 ARITHMETIC EXPRESSIONS

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

a. a constant

b. a variable

c. an array element

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

e. an arithmetic or logical expression written within parentheses.

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

The following are valid examples of simple arithmetic expressions:

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

A compound arithmetic expression consists of two or more operands combined by arithmetic operators. The arithmetic operations permitted in FORTRAN-10 and the operator recognized for each are given in Table 4-1.

Table 4-1
Arithmetic Operations and Operators

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

4.1.1 Rules for Writing Arithmetic Expressions

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

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

NOTE
Only one combination of data types, double precision with complex, is prohibited in FORTRAN-10.

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

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

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

4.2 LOGICAL EXPRESSIONS

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

a. constant

b. variable

c. array element

d. function reference (see Chapter 15), or

e. another expression written within parentheses.

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

The logical operators permitted by FORTRAN-10 and a description of the operation each provides are given in Table 4-4.

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

Type of Argument 2

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

Type of Argument 1

Table 4-3
Permitted Base/Exponent Type Combinations

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

Table 4-4
Logical Operators

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

Logical expressions are written in the general form P_1 .OP. P_2 where P is a logical operand and .OP. is any logical operator but .NOT. The .NOT. operator complements the value of a logical operand and must be written immediately before the operand which it modifies (e.g., .NOT.P). A truth table illustrating all possible logical combinations of two logical operands (P and Q) and the resultant of each combination is given in Table 4-5.

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

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

Table 4-5
Logical Operations, Truth Table

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

Examples

Assume the following variables:

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

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

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

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

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

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

Table 4-6
Binary Logical Operations, Truth Table

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

4.2.1 Relational Expressions

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

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

In FORTRAN-10, relational operators may be written either as a two-letter mnemonic enclosed within periods (e.g., .GT.) or *symbolically* using the *symbols* >, <, = *and* #. Table 4-7 lists both the mnemonic and symbolic forms of the FORTRAN-10 relational operators and specifies the type of quantitative test performed by each operator.

Table 4-7
Relational Operators and Operations

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

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

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

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

Examples

Assume the following variables:

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

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

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

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

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

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

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

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

4.3 EVALUATION OF EXPRESSIONS

The order of computation of a FORTRAN-10 expression is determined by

a. the use of parentheses

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

c. the location of operators within an expression.

4.3.1 Parenthesized Subexpressions

In an expression all subexpressions written within parentheses are evaluated first. When parenthesized subexpressions are nested (one contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second and so on, until the value of the final parenthesized expression is computed. When more than one operator is contained by a parenthesized subexpression, the required computations are performed according to the hierarchy assigned operators by FORTRAN-10 (Paragraph 4.3.2).

Example:

The separate computations performed in evaluating the expression

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

a. A/B = *R1*

b. *R1*+C = *R2*

c. B/*R2* = *R3*

d. *R3*-C = *R4*

e. A+*R4* = *R5*

NOTE

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

4.3.2 Hierarchy of Operators

The following hierarchy (i.e., order of execution) is assigned to the classes of FORTRAN-10 operators:

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

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

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

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

Table 4-8
Hierarchy of FORTRAN-10 Operators

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

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

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

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

4.3.3 Mixed Mode Expressions

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

Example

Assume the following:

Variable	Type
D	Double Precision
X	Real
I,J	Integer

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

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

a.	(I/J) = R1	R1 is integer
b.	X*R1 = R2	R1 is converted to type real and is multiplied by X to produce R2
c.	D+R2 = R3	R2 is converted to type double precision and is added to D to produce R3

4.3.4 Use of Logical Operands in Mixed Mode Expressions

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 5
COMPILATION CONTROL STATEMENTS

5.1 INTRODUCTION

Compilation control statements are used to identify FORTRAN-10 programs and to specify their termination. Statements of this type do not affect either the operations performed by the object program or the manner in which the object program is executed.

5.2 END STATEMENT

This statement is used to signal FORTRAN-10 that the physical end of a source program or subprogram has been reached. END is a nonexecutable statement. The general form of an END statement is

```
END
```

The following rules govern the use of the END statement:

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

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

c. An END statement cannot be labeled.

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 6
SPECIFICATION STATEMENT

6.1 INTRODUCTION

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

a. DIMENSION

b. Statements which specify, explicitly, type.

c. *IMPLICIT*

d. COMMON

e. EQUIVALENCE

f. EXTERNAL

Specification statements are nonexecutable and, with the exception of ***IMPLICIT,*** may appear anywhere in the source program. They must, however, precede any executable statement that references variables which they specify.

6.2 DIMENSION STATEMENT

DIMENSION statements provide FORTRAN-10 with information needed to identify and allocate the space required for source program arrays. Any number of subscripted array names may be specified as array declarators in a DIMENSION statement. The general form of a DIMENSION statement is

DIMENSION S1, S2, . . . , Sn

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

name *(min/max,min/max,. . .,min/max)*

where *name* is the symbolic name of the array and each ***min/max subscript quantity represents minimum and maximum values of an array dimension.***

Each min/max value for an array dimension may be either an integer constant or an integer variable. The value given the minimum specification for a dimension must not exceed the value given the maximum specification. Minimum values of 1 with their following slash delimiter may be omitted from a dimension subscript.

Examples

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

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

6.2.1 Adjustable Dimensions

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

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

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

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

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

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

Examples

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

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

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

Note that for the calling program of the form:

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

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

6.3 TYPE SPECIFICATION STATEMENTS

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

Type specification statements are written in the following form:

```
type V1, V2, . . .,Vn
```

where type may be any one of the following declarators:

a. INTEGER

b. REAL

c. DOUBLE PRECISION

d. COMPLEX

e. LOGICAL

NOTE

In order to be consistent with the type statements used by other manufacturers, the following type declarators are also accepted by FORTRAN-10:

Declarator	Form of Variable Specified
*INTEGER*4*	*Full word integer*
*REAL*4*	*Full word real*
*REAL*8*	*Double Precision (real)*

The type statement list (i.e., V1, V2, . . .,Vn) consists of any number of variable, array, or function names which are to be declared the specified type. The names listed must be separated by commas; a name may appear in only one type statement in the same source program.

Examples

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

NOTE

Variables, arrays, and functions of a source program, which are not typed either implicitly or explicitly by a specification statement, are typed by FORTRAN-10 according to the following conventions:

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

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

6.4 IMPLICIT STATEMENTS

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

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

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

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

where type represents a data type name and the parenthesized list represents a list of different letters. The data type names given can be

a. INTEGER

b. REAL

c. DOUBLE PRECISION

d. COMPLEX or

e. LOGICAL.

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

NOTE

Type declarations given in an explicit type specification override those also given in an IMPLICIT statement. IMPLICIT declarations do not affect the FORTRAN-10 supplied functions.

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

When used, an IMPLICIT statement must appear before any other declaration statement in the program or subprogram in which it appears. IMPLICIT statements may be used more than once, but conflicting type statements should not be given.

6.5 COMMON STATEMENT

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

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

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

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

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

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

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

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

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

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

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

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

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

During compilation of a source program, FORTRAN-10 will string together all items listed for each labeled common area and for blank common in the order in which they appear in the source program statements. For example, the series of source program statements

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

have the same effect as the single statement

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

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

The largest common area must be loaded first.

6.5.1 Dimensioning Arrays in COMMON Statements

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

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

Example

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

6.6 EQUIVALENCE STATEMENT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
↑ A(1)
| A(2)
X A(3)
Y A(4)
Z
```

6.7 EXTERNAL STATEMENT

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

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

where each name listed is declared to be a subprogram name.

If a subprogram is given the same name as a FORTRAN-10 library function, that name must be declared a subprogram name in an EXTERNAL statement.

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 7
DATA STATEMENT

7.1 INTRODUCTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In instances where the type of the data specified is not the same as that of the variable to which it is assigned. FORTRAN-10 converts the datum to the type of the variable. The type conversion is performed using the rules given for type conversion in arithmetic assignments (refer to Chapter 8, Table 8-1). Octal, logical, and literal constants are not converted.

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

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

Example

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

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

will cause

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

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

Example

Assuming that the variable C is complex, the statement

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

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 8
ASSIGNMENT STATEMENTS

8.1 INTRODUCTION

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

a. Arithmetic assignment statements

b. Logical assignment statements

c. Statement Label assignment (ASSIGN) statements.

8.2 ARITHMETIC ASSIGNMENT STATEMENT

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

v=e

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

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

When the type of the specified variable or array element name differs from that of its assigned value, FORTRAN-10 converts the value of the type of its assigned variable or array element. The type conversion operations performed by FORTRAN-10 for each possible combination of variable and value types are described in Table 8-1.

Table 8-1
Rules for Conversion in Mixed Mode Assignments

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

Legend

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

Notes

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

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

*** Use the first word of the literal.

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

8.3 LOGICAL ASSIGNMENT STATEMENTS

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

v=e

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

Examples

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

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

8.4 ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT

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

ASSIGN n TO I

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

ASSIGN 2000 TO LABEL

specifies that the variable LABEL represents the statement number 2000.

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

Once a variable has been assigned a statement number, FORTRAN-10 will consider it a label variable. If a label variable is used in an arithmetic statement, the result will be unpredictable.

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

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 9
CONTROL STATEMENTS

9.1 INTRODUCTION

FORTRAN-10 object programs are normally executed statement-by-statement in the order in which they were presented to the compiler. The following source program control statements, however, enable the user to alter the normal sequence of statement execution:

a. GO TO

b. IF

c. DO

d. CONTINUE

e. STOP

f. PAUSE

9.2 GO TO CONTROL STATEMENTS

There are three kinds of GO TO statements:

a. Unconditional

b. Computed

c. Assigned.

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

9.2.1 Unconditional GO TO Statements

GO TO statements of this type are written in the form

```
GO TO n
```

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

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

9.2.2 Computed GO TO Statements

GO TO statements of this type are written in the form

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

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

NOTE
A comma may optionally follow the parenthesized list.

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

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

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

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

9.2.3 Assigned GO TO Statements

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

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

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

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

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

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

Examples

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

9.3 IF STATEMENTS

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

9.3.1 Arithmetic IF Statements

IF statements of this type are written in the form

IF (E) L1, L2, L3

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

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

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

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

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

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

Examples

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

9.3.2 Logical IF Statements

IF statements of this type are written in the form

IF (E) S

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

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

The statement given in a logical IF statement may be any FORTRAN-10 executable statement except a DO statement or another logical IF statement.

Examples

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

9.3.3 Logical Two-Branch IF Statements

IF statements of this type are written in the form

IF (E) N1, N2

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

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

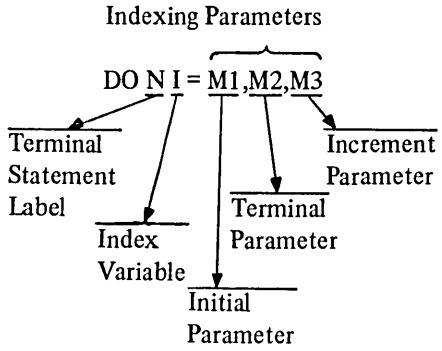
Note that the statement immediately following the logical two-branch IF must be numbered so that control can later be transferred to the portion of code that was skipped.

Examples

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

9.4 DO STATEMENT

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



where

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

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

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

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

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

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

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

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

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

(M2-M1)/M3+1

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

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

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

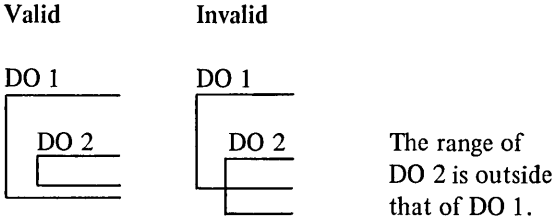
Exit from a DO loop may also be accomplished by a transfer of control by a statement within the DO loop range to a statement outside the range of the DO statement (Paragraph 9.4.3).

9.4.1 Nested DO Statements

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

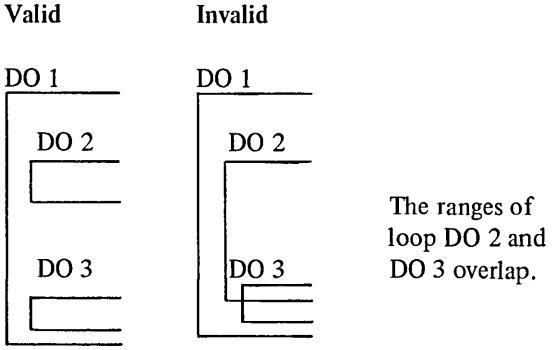
a. The range of each nested DO statement must be entirely within the range of the containing DO statement.

Example



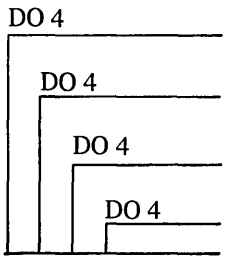
b. The ranges of nested DO statements cannot overlap.

Example



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

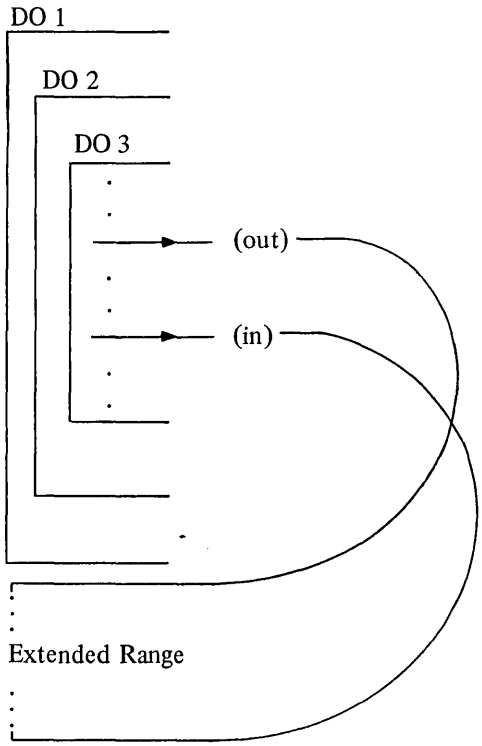
Example



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

9.4.2 Extend Range

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



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

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

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

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

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

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

9.4.3 Permitted Transfer Operations

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

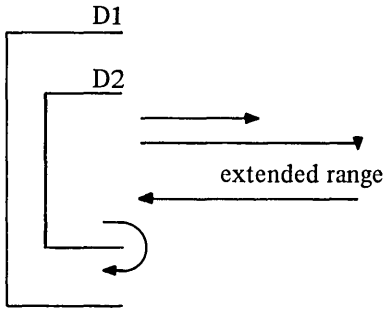
a. A transfer out of the range of any DO loop is permitted at any time. When such a transfer is executed the value of the controlling DO statement's index variable is defined as the current value.

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

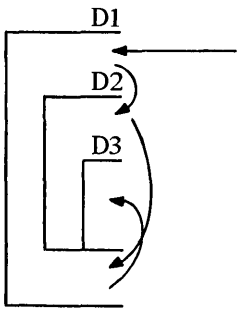
c. The use of and return from a subprogram from within the range of any DO loop, nested DO loop, or extended range is permitted.

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

Valid Transfers



Invalid Transfers



9.5 CONTINUE STATEMENT

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

```
12    CONTINUE
```

Example

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

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

9.6 STOP STATEMENT

When executed, the STOP statement causes the execution of the object program to be terminated and control returned to the DECsystem-10 Monitor. A descriptive message may, optionally, be included in the STOP statement to be output to the user's I/O terminal immediately before program execution is terminated. This statement may be written as

```
STOP
STOP 'N'
```

or

```
STOP n
```

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

Examples

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

or

```
STOP 7777
```

9.7 PAUSE STATEMENT

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

a. Continue execution of the program

b. Exit

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

The permitted forms of the PAUSE statement are:

a. PAUSE

b. PAUSE *'literal string'*

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

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

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

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

PAUSE *'TEST POINT A'*

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

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

The statement

PAUSE 1

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

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

9.7.1 T (TRACE) Option

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

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

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

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

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

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

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

Example

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

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

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

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

CALL TRACE

or as a function, using the form

X = TRACE (x)

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 10
I/O STATEMENTS

10.1 DATA TRANSFER OPERATIONS

FORTRAN-10 I/O statements permit data to be transferred between processor storage (core) and peripheral devices and/or between storage locations. Data in the form of logical records may be transferred using an a) sequential, b) ***random access,*** or c) ***append*** transfer mode. The areas in core from which data is to be taken during output (write) operations and into which data is stored during input (read) operations are specified by

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

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

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

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

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

10.2 TRANSFER MODES

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

10.2.1 Sequential Mode

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

10.2.2 Random Access Mode

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

The FORTRAN-10 OPEN statement or a subroutine call to DEFINE FILE may be used to set up random access files.

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

10.2.3 Append Mode

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

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

10.3 I/O STATEMENTS, BASIC FORMATS AND COMPONENTS

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

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

where

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

u = FORTRAN-10 logical unit number

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

list = I/O list

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

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

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

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

10.3.1 I/O Statement Keywords

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

a. READ

b. REREAD

c. WRITE

d. ACCEPT

e. PRINT

f. PUNCH

g. TYPE

h. FIND

i. ENCODE

j. DECODE

10.3.2 FORTRAN-10 Logical Unit Numbers

The physical devices used for most FORTRAN-10 I/O operations are identified by decimal numbers. During compilation, the compiler assigns default logical unit numbers for the ***REREAD,*** READ, ***ACCEPT,*** PRINT, PUNCH, and ***TYPE*** statements. Default unit numbers are negatively signed decimal numbers that are inaccessible to the user.

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

10.3.3 FORMAT Statement References

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

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

Unformatted files are binary files divided into records by FORTRAN-10 embedded control words; the control words are invisible to the user. Files of this type cannot be prepared by the user without utilizing FOROTS. Unformatted files are intended to be used only within the FORTRAN-10 environment.

Table 10-1
FORTRAN-10 Logical Device Assignments

Device/Function	*Default Filename*	*FORTRAN Logical Unit Number*	*Use*
*Standard Devices**			
0	*FORxx.DAT*	*00*	*ILLEGAL*
DSK		*01*	*DISK*
CDR		*02*	*Card Reader*
LPT		*03*	*Line Printer*
CTY		*04*	*Console Teletype*
TTY		*05*	*User's Teletype*
PTR		*06*	*Paper Tape Reader*
PTP		*07*	*Paper Tape Punch*
DIS		*08*	*Display*
DTA1		*09*	*DECtape*
DTA2		*10*	
DTA3		*11*	
DTA4		*12*	
DTA5		*13*	
DTA6		*14*	
DTA7		*15*	*DECtape*
MTA0		*16*	*Magnetic Tape*
MTA1		*17*	
MTA2		*18*	
FORTR		*19*	*Assignable Device*
DSK		*20*	*DISK*
DSK		*21*	
DSK		*22*	
DSK		*23*	
DSK		*24*	
DEV1		*25*	*Assignable Devices*
DEV2		*26*	
DEV3		*27*	
DEV4		*28*	
DEV5		*29*	
DEV63	*FOR63.DAT*	*63*	*DISK*
Default Devices (inaccessible to the user)			
REREAD	*Current file*	*-6*	*REREAD statement*
CDR	*CDR.DAT*	*-5*	*READ statement*
TTY	*TTY.DAT*	*-4*	*ACCEPT statement*
LPT	*LPT.DAT*	*-3*	*PRINT statement*
PTP	*FORPTP.DAT*	*-2*	*PUNCH statement*
TTY	*FORTTY.DAT*	*-1*	*TYPE statement*

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

10.3.4 I/O List

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

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

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

The general form of an implied DO is

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

where

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

Examples

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

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

Example

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

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

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

Example

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

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

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

10.3.5 The Specification of Records for Random Access

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

NOTE

A pound sign (#) may be used in place of the ' delimiter (e.g., both #101 and '101 are accepted by FORTRAN-10).

10.3.6 List-Directed I/O

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

keyword (u,)list*

Example

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

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

NOTE

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

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

a. Input constants must be of a type acceptable to FORTRAN-10. Octal constants, although acceptable, are not permitted in list-directed I/O operations.

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

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

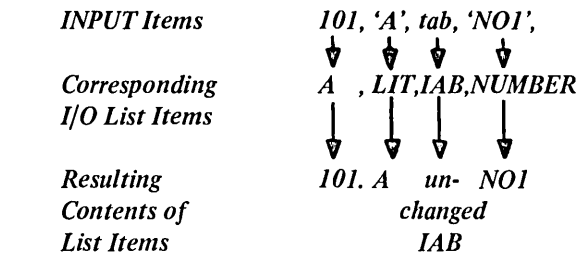
d. *Decimal points may be omitted from real constants which do not have a fractional part. FORTRAN-10 assumes that the decimal point follows the right-most digit of a real constant.*

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

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

b. *Delimiters may be either preceded by or followed by any number of blanks, carriage return/line feed characters, tabs, or line terminators; any such combination is considered by FORTRAN-10 as being only a single delimiter.*

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



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

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

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

The repetition of a constant is written as

*r*K*

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

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

Examples

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

10.3.7 NAMELIST I/O Lists

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

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

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

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

transfers program control to statement 50.

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

10.5 READ STATEMENTS

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

10.5.1 Sequential Formatted READ Transfers

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

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

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

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

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

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

Example: READ(15,101)

c. **Form:** READ f

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

Example: READ 66

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

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

Example: READ 15, ARRAY (20,30)

10.5.2 Sequential Unformatted Binary READ Transfers

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

Form: READ (u,)list

Use: Input one logical record of data from logical unit *u* into processor storage as the value of the location identified in list. Only binary files that have been output by a FORTRAN-10 unformatted WRITE statement may be read by this type of READ statement.

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

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

10.5.3 Sequential List-Directed READ Transfers

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

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

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

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

b. Form: *READ *, list*

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

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

10.5.4 Sequential NAMELIST-Controlled READ Transfers

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

Form: *READ (u,n)*

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

10.5.5 Random Access Formatted READ Transfers

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

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

Use: *Input data from record R of logical unit u. Format each input datum according to the format specifications of f and place into processor storage as values of the locations identified in list. Only disk files that have been set up by either an OPEN or DEFINE FILE statement may be accessed by a READ statement of this form.*

10.5.6 Random Access Unformatted READ Transfers

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

Form: *READ (u#R)list*

Use: *Input data from record R of logical unit u. Place the input data into processor storage as the value of the locations identified in list. Only binary files that have been output by an unformatted random-access WRITE statement may be accessed by a READ statement of this form.*

Example: *READ (01#20) BINFIL*

Read record number 20 into array BINFIL.

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

10.6 SUMMARY OF READ STATEMENTS

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

Table 10-2
Summary of Read Statements

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

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

10.7 REREAD STATEMENT

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

The REREAD feature of FORTRAN-10 cannot be used until an input (READ) transfer from a file has been accomplished. If REREAD is used prematurely, an error message will be output by FORTRAN-10 at execution time.

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

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

Form: *REREAD f,list*

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

Example:

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

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

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

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

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

10.8 WRITE STATEMENTS

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

10.8.1 Sequential Formatted WRITE Transfers

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

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

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

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

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

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

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

c. **Form:** WRITE f

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

Example: WRITE 10

10.8.2 Sequential Unformatted WRITE Transfer

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

Form: WRITE (u) list

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

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

10.8.3 Sequential List-Directed WRITE Transfers

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

Form: *WRITE(u,*)list*

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

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

10.8.4 Sequential NAMELIST-Controlled WRITE Transfers

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

Form: *WRITE(u,N)*

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

Example: *WRITE(12,NMLST)*

10.8.5 Random Access Formatted WRITE Transfers

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

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

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

10.8.6 Random Access Unformatted WRITE Transfers

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

Form: *WRITE(u#R)list*

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

10.9 SUMMARY OF WRITE STATEMENTS

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

Table 10-3
Summary of WRITE Statements

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

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

10.10 ACCEPT STATEMENT

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

10.10.1 Formatted ACCEPT Transfers

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

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

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

Example: *ACCEPT 101,LINE(73)*

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

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

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

10.10.2 ACCEPT Transfers Into FORMAT Statement

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

Form: *ACCEPT f*

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

Example: *ACCEPT 101*

10.11 PRINT STATEMENT

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

a. Form: PRINT f,list

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

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

b. Form: PRINT *,list

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

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

c. Form: PRINT f

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

Example: PRINT 55

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

Example

The sequence

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

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

10.12 PUNCH STATEMENT

The PUNCH statement causes data from specified processor storage locations to be output to the system's standard paper tape punch (PTP). (See Table 10-1 for device assignments.) This statement may be used only for sequential formatted data transfers and may be written in one of the three following forms:

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

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

   Example: PUNCH 10,TABLE(10,20),I,J,K

b. **Form:** PUNCH *,list

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

   Example: PUNCH *,I,A,B,M,TAB(5,10)

c. **Form:** PUNCH f

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

The latter form of the PUNCH statement is particularly useful when employed in conjunction with an ACCEPT f statement to cause user-entered data (i.e., comments or headings) to be added to an output file at program execution time.

10.13 TYPE STATEMENT

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

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

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

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

b. *Form:* *TYPE f*

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

Example: *TYPE 101*

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

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

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

10.14 FIND STATEMENT

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

The form of the FIND statement is

```
FIND (u#R)
```

Example

In the sequence

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

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

10.15 ENCODE AND DECODE STATEMENTS

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

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

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

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

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

where

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

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

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

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

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

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

10.15.1 ENCODE Statement

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

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

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

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

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

10.15.2 DECODE Statement

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

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

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

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

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

10.15.3 Example of ENCODE/DECODE Operations

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

Example

Assume the contents of the variables to be as follows:

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

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

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

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

The result after the second iteration is:

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

The DECODE statement

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

b. *converts them to floating point binary value*

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

d. *skips the next character*

e. *extracts the digit 5 from C*

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

g. *stores it in B(4).*

10.16 SUMMARY OF I/O STATEMENTS

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

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

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

Legend:

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

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 11
NAMELIST STATEMENTS

11.1 INTRODUCTION

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

11.2 NAMELIST STATEMENT

NAMELIST statements are written in the following form:

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

where

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

Example

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

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

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

The rules for structuring a NAMELIST statement are:

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

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

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

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

11.2.1 NAMELIST-Controlled Input Transfers

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

V=K1,K2,. . .,Kn

where

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

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

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

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

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

11.2.2 NAMELIST-Controlled Output Transfers

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

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

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

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

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

generates the following form of output:

```
Column

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 12
FILE CONTROL STATEMENTS

12.1 INTRODUCTION

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

The OPEN, CLOSE, and CALL RELEASE statements are described in this chapter.

12.2 OPEN AND CLOSE STATEMENTS

Both the OPEN and CLOSE statements are unique to FORTRAN-10; they both use the same format and have the same options and arguments.

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

a. the name and location of the data file

b. the type of access required

c. the data format within the file

d. the protection code[1] to be assigned an output data file

e. the disposition of the data file

f. data file record, block and file sizes

g. a data file version identifier

h. error modes

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

---

[1] *Refer to Chapter 6 of the DECsystem-10 Monitor Calls Manual, DEC-10-MRRC-D, for a description of file access protection codes.*

The general form of the OPEN statement is:

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

The CLOSE statement is used in the termination of an I/O operation to dissociate the I/O device being used from the active file and file-related information, and to restore the core occupied by I/O buffers and other transfer-related operations. All required device dependent termination functions are also performed on the execution of a CLOSE statement; the I/O device, however, is not released.

Once a CLOSE statement has been executed, another OPEN statement is required to regain access to the closed file.

The general form of the CLOSE statement is:

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

12.2.1 Options for OPEN and CLOSE Statements

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

a. UNIT

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

UNIT = An integer variable or constant

NOTE

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

b. DEVICE

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

DEVICE = A literal constant or variable

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

c. ACCESS

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

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

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

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

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

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

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

The general form of the ACCESS argument is:

ACCESS = *'SEQIN'* | *'SEQOUT'* | *'SEQINOUT'* | *'RANDOM'* | *'RANDIN'* | *'APPEND'*

d. *MODE*

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

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

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

NOTE

Refer to the DECsystem-10 Monitor Calls Manual for a detailed description of the data modes given in the following list.

Literal	*Action Indicated*
'ASCII'	*Specifies an ASCII character set.*
'BINARY'	*Specifies data formatted as a FORTRAN binary data file.*
'IMAGE'	*Specifies an image (I) mode data transfer for the associated READ or WRITE statements. IMAGE is an unformatted binary mode.*
'DUMP'	*The data file to be transferred is to be handled in a DUMP mode of operation.*

The general form of the MODE argument is:

MODE = *'ASCII'* / *'BINARY'* / *'IMAGE'* / *'DUMP'*

e. *DISPOSE*

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

Literal	*Action Indicated*
'SAVE'	*Leave the file on the device.*
'DELETE'	*If the device involved is either a DECtape or disk, remove the file; otherwise, take no action.*
'PRINT'	*If the file is on disk, queue it for printing; otherwise, take no action.*
'PUNCH'	*Paper tape punch output.*
'RENAME'	*Change filename.*

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

DISPOSE = *'SAVE'* / *'DELETE'* / *'PRINT'* / *'PUNCH'* / *'RENAME'*

f. FILE

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

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

FORxx.DAT

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

FILE =	*An ASCII literal, a complex, single precision, or double precision variable.*

g. PROTECTION

This option specifies a protection code to be assigned the data file being transferred. The protection code determines the level of access to the file that three possible classes of users (i.e., owner, member, or other) will have. PROTECTION may be a 3-digit octal literal or a variable; if the argument is assigned a zero value or is not given, the default protection code established for the DECsystem-10 installation is used. The general form of the PROTECTION argument is:

PROTECTION =	*3-digit octal, a literal, or variable*

h. DIRECTORY

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

1. *the user's project programmer number which identifies the UFD, for example, 10,7, or*

2. *A UFD/SFD directory path specification. A path specification lists the UFD and the names of its SFD's which form a path to the desired SFD. For example, the following path specification identifies the path leading to SFD 1234:*

10,7,SFDA,SFDB,1234

NOTE

Refer to the DECsystem-10 Monitor Calls manual for a complete description of directories and multilevel directory structures.

The general form of a DIRECTORY argument is:

DIRECTORY= UFD name or directory path specification

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

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

1. Single Precision Array

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

where PATH and its elements are:

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

2. Double Precision Array

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

where PATH and its elements are:

```
DOUBLE PRECISION PATH (5)

PATH (1) "000010000007 – – – (PROJ, PROG. NUMBERS
                               = UFD)
PATH (2) 'SFDABC'  ]
PATH (3) 'MYAREA'  ]  names of sub-file directories (SFD's)
PATH (4) 'YOURIT'  ]
PATH (5) 0
```

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

The following is an example of a literal specification:

```
(DIRECTORY='10,7,SFD1,SFD2,SFD3'
            |   |   |
            |   |   Sub-File
            |   |   Directory
            |   |   Path
            |   Programmer
            |   Number
            Project
```

Whenever the specification is an array, the required project and programmer numbers may be specified either as one word with the project number in the left half and the programmer number in the right half, or as the right halves of separate sequential word locations.

i. BUFFER COUNT

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

BUFFER COUNT = An integer constant or variable

j. FILE SIZE

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

FILE SIZE = An integer constant or variable

k. VERSION

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

VERSION = An octal constant or variable

l. BLOCK SIZE

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

BLOCK SIZE = An integer, constant or variable

m. RECORD SIZE

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

RECORD SIZE = An integer constant or variable

n. ASSOCIATE VARIABLE

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

ASSOCIATE VARIABLE = Variable Name

o. PARITY

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

PARITY = 'ODD'
'EVEN'

p. DENSITY

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

DENSITY = '200'
'556'
'800'

q. DIALOG

The use of this option in an OPEN statement enables the user to supersede or defer, at execution time, the values previously assigned to the arguments of the statement. The value assigned to DIALOG may be null, a literal, or an array. The general form of this argument is:

DIALOG = Literal, array, or null

Whenever DIALOG is assigned a null value, it establishes a user/program dialogue mode when the OPEN statement containing it is executed. During a dialogue mode FOROTS outputs the following messages at the user's terminal.

ENTER FILE SPECIFICATIONS FOR LOGICAL UNIT XX

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

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

Whenever a literal or an array is assigned to DIALOG, it must contain, in ASCII, the file specification information or indicate where to request dialog information.

12.2.2 Summary of OPEN/CLOSE Statement Options

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

Table 12-1
OPEN/CLOSE Statement Arguments

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

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 13
FORMAT STATEMENT

13.1 INTRODUCTION

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

FORMAT statements may appear anywhere in a FORTRAN-10 source program; however, they must be labeled to enable them to be referenced by I/O statements.

13.1.1 FORMAT Statement, General Form

The general form of a FORMAT statement follows:

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

where

k = the required statement number

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

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

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

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

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

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

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

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

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

may be written as

FORMAT(3(SA1,SA2,SA3))

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

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

may also be written in the form

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

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

13.2 FIELD DESCRIPTORS

FORMAT statement field descriptors describe the desired conversion, scaling, and editing of data for a specific field within a record. In FORTRAN-10, no restrictions are placed on the depth to which parenthesized format field descriptors may be nested. FORTRAN-10 permits the following forms of field descriptors:

Forms	Comments
rFw.d rEw.d rDw.d rGw.d	Floating point numeric field descriptors
rIw	Integer field descriptor
rLw	Logical field descriptor
rAw	Alphanumeric data field descriptor
nHs	Alphanumeric data in a FORMAT statement field descriptor
nX *Tw*	Formatting field descriptors
nP	Numerical scale factor descriptor

Forms	Comments
/	Record delimiter
$	Format in field descriptor
's'	A string of ASCII characters within a FORMAT statement
rOw	Octal field descriptor

where

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

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

.d = an optional unsigned integer that specifies the number of fractional digits which are to appear in the external representation of the field being described.

r = an unsigned integer that specifies the number of characters to be processed during the transfer of alphanumeric data or formatting character counts or a signed scale factor.

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

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

The characters F,E,D,G,I,L,A,H,X,T,*O,* and P indicate the manner of conversion and editing to be performed between the internal (processor) and external representations of the data within a specific field; these characters are referred to as conversion codes. The FORTRAN-10 conversion codes and a brief description of the function of each are given in Table 13-1.

Table 13-1
FORTRAN-10 Conversion Codes

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

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

Detailed descriptions of the various types of field descriptors, the manner in which they are written and employed and their use in FORMAT statements are given in the following paragraphs.

13.2.1 Numeric Field Descriptors

The forms of the field descriptors used to specify the format and conversion of numeric data follow.

Description	Type of Data Used For
Dw.d	Double precision real data with a D exponent
Ew.d	Real data with an E exponent
Ew.d,Ew.d	For the real and imaginary parts of a complex datum
Fw.d	Real data without an exponent
Fw.d,Fw.d	For the real and imaginary parts of a complex datum
Iw	Integer data
Ow	Octal data
Gw.d	Real or double precision data
Gw	For integer (or logical) data
Gw.d,Gw.d	For the real and imaginary parts of a complex datum

NOTE

The G conversion code may be used for all but octal numeric data types.

Examples

Consider the following program segment:

```
INTEGER I1, I2
REAL R1, R2, R3
DOUBLE PRECISION D1, D2
I1 = 506
I2 = 8
R1 = 506.0
R2 = 18.1
R3 = 506001.0
D1 = 18.0
D2 = -504.0
.
.
.
```

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

Table 13-2
Action of Field Descriptors On Sample Data

Item	Descriptor Form	Sample Descriptor	WRITE Statement Using the Sample Descriptor	External Form of Sample Field Described	External Appearance of Sample Data
1	Dw.d	D8.2	WRITE (-,-) D1	Z.nnD±nn	0.18D+02
2	Ew.d	E8.2	WRITE (-,-) R1	Z.nnE±nn	0.51E+03
3	Fw.d	F5.2	WRITE (-,-) R2	aa.nn	18.10
4	Iw	I5	WRITE (-,-) I1	aaaan	␣␣506
5	Iw	I2	WRITE (-,-) I1	an	**
6	Ow	O5	WRITE (-,-) I2	nnnnn	00010
7	Gw.d	G8.2	WRITE (-,-) D2	Z.nnD±nn	-.50D+02
8	Gw.d	G8.2	WRITE (-,-) R3	Z.nnE±nn	0.51E+06
9	Gw.d	G8.2	WRITE (-,-) R2	aa.nn	18.10
10	Gw	G5	WRITE (-,-) I1	aaaan	␣␣506

where: a. *n* represents a numeric character

b. *Z* represents either a - or 0 (Note that if n-d > 6, a negative number cannot be output.)

c. *a* represents a digit, leading blank (␣) or a minus sign depending on the numeric output.

Notes:

1. In Item 1, the value D1 has only 2 significant digits and d=2, so no rounding will occur on input.
2. In Item 2, since R1 has 3 significant digits, it is rounded to fit into the specified field.
3. In Item 5, the width (*w*) part of a format descriptor specifies an exact field which permits no rounding of its contents. If the *w* specification is too small for the datum to be transferred, asterisks are output to indicate that the transfer was not made.
4. In Item 6, Integer 8 = Octal 10.
5. In Items 8 and 9, the relationship between G and fixed and floating real data is discussed in Paragraph 13.2.3.

The internal and external forms of the data specified by the numeric format conversion code are summarized in Table 13-3.

Table 13-3
Numeric Field Codes

Internal Form	Conversion Code	External Form
Binary floating point double precision	D	Decimal floating point with D exponent
Binary floating point	E	Decimal floating point with E exponent
Binary floating point	F	Decimal fixed point
Binary integer	I	Decimal integer
Binary word	*O*	*Octal value*
One of the following: single precision, binary floating point, binary integer, binary logical, or binary complex	G	Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form

Complex quantities are transferred as two independent real quantities. The format specification for complex quantities consists of either two successive real field descriptors or one repeated real field descriptor. For example, the statement

```
FORMAT(2E15.4,2(F8.3,F8.5))
```

may transfer up to three complex quantities.

The equivalent of the foregoing statement is

```
FORMAT(E15.4,E15.4,F8.3,F8.5,F8.3,F8.5)
```

13.2.2 Interaction of Field Descriptors With I/O List Variables During Transfer

The execution of an I/O statement that specifies a formatted data transfer operation initiates format control. The actions performed by format control depend on information provided by the elements of the I/O statement's list of variables and the field descriptors which comprise the referenced FORMAT statement's format specifications.

In processing each FORMAT controlled I/O statement which has an I/O list, FORTRAN-10 scans the contents of the list and the format specifications in step. Each time another variable or array element name is obtained from the list, the next field specification is obtained from the format specification. If the end of the format specification is reached and more items remain in the list, a new line or record is established and the scan process is restarted, either at the first item in the format specification or, if parenthesized sets of format specifications exist within the format specification, with the last set within the format specification.

During transfer operations the corresponding list variables and field descriptors (i.e., first variable and first field descriptor, second variable and second field descriptor, etc.) are used by format control to determine which datum to transfer and its final form on completion of the transfer.

During formatted read operations, the first input record is read when format control is initiated, additional records are then read only when specified by the format specifications.

During formatted write operations, the writing of a record occurs each time one is specified in the format specification.

In both the formatted read and write operations all unprocessed characters of a record (input or output) are skipped when

a. a slash (/) is found in the format specification,

b. the delimiting right parentheses,), of the FORMAT statement is found and

c. if there are no more items in the input list or Hollerith field descriptors in the FORMAT statement.

When the scan of the format specification reaches the right closing parenthesis, format control tests to see if any variable remains in the I/O statement list. If no list variable is found, format control is terminated and the program proceeds to the next statement. If a list variable is found, transfer operations are continued starting with the first field descriptor of the format specification or, if uncounted repeated groups of specifications exist within the format specification, to the last (rightmost) uncounted repeat specification group. In either case, a new record is accessed.

13.2.3 G, General Numeric Conversion Code

The G conversion code may be used in field descriptors for the format control of real, double precision, integer, logical, or complex data.

With the exception of real and double precision data, the type of conversion performed by a G type field descriptor depends on the type of its corresponding I/O list variable. In the case of real and double precision data, the kind of conversion performed is a function of the external magnitude of the datum being transferred. Table 13-4 illustrates the conversions performed for various ranges of magnitude (external form) of real and double-precision data.

13.2.4 Numeric Fields with Scale Factors

Scale factors may be added to D,E,F, and G conversion codes in field descriptors. The scale factor has the form

nP

where *n* is a signed integer (+ is optional) and P identifies the operation. When used, a scale factor is added as a prefix to field descriptors.

Examples

-2PF10.5
1PE8.2

When added to an F type field descriptor (or G type if the external field is a fixed point decimal) a scale factor specifies a power of 10 so that

External Form of Number = (Internal Form)*$10^{(\text{scale factor})}$

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

F8.3

produces the external field

␣␣26.451

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

Magnitude of Data in its External Form (M)	Equivalent Method of Conversion Performed
$0.1 \leqslant M < 1$	F(w-4).d,4X
$1 \leqslant M < 10$	F(w-4).(d-1),4X
.	.
.	.
.	.
.	.
$10^{d-2} \leqslant M < 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leqslant M < 10^{d}$	F(w-4).0,4X
ALL OTHERS	Ew.d

Note: In all numeric field conversions the field width (w) specified should be large enough to include the decimal point, sign, and exponent character in addition to the number of digits. If the specified width is too small to accommodate the converted number, the field will be filled with asterisks (*). If the number converted occupies fewer character positions than specified by *w*, it will be right-justified in the field and leading blanks will be used to fill the field.

The addition of the scale factor of -1P

```
-1PF8.3
```

produces the external field

```
bbb2.645
```

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

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

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

13.2.5 Logical Field Descriptors

Logical data may be transferred under format control in a manner similar to numeric data transfer by use of the field descriptor

Lw

where *L* is the control character and *w* is an integer specifying the field width. The data is transmitted as the value of a corresponding logical variable in the associated input/output list.

If, on input, the first nonblank character in the logical data fields is T or F, the value of the logical variable is stored in the list variable as true or false, respectively. If the entire input data field is blank or empty, a value of false is stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

13.2.6 Variable Numeric Field Widths

The D, E, F, G, and I conversion codes may appear in a FORMAT statement without the w (field width) or d (the number of places after the decimal point) portion of the w.d specification. In the case of input, omitting *w* implies that the numeric field is delimited by the first character which is illegal in the field, or one of the characters -, +, ., E, D, or blank, provided it is not the first character of the numeric field. For example, input according to the format

10 FORMAT(2I,F,E)

might appear on the input medium as

-10,3/15.621-.0016E-10,777

If *w* is given and *d* is not, *d* is assumed to be zero. During data output operations if *d* and *w* are omitted[1], the following defaults are used:

Format Code	Assumed Default for KA10	Assumed Default for KI10
D	D25.16	D25.18
E	E15.7	E15.7
F	F15.7	F15.7
G single precision	G15.7	G15.7
G double precision	G25.16	G25.18
I	I15	I15
L	L15	L15

13.2.7 Alphanumeric Field Descriptors

The formatted transfer of alphanumeric data may be accomplished in a manner similar to the formatted transfer of numeric data by use of the field descriptor Aw, where *A*, is the control character and *w* is the number of characters

[1] If *d* is given and *w* is left out, *d* is ignored and the default is used.

in the field. Alphanumeric characters are transferred into or from a variable in an input/output list depending on the I/O operation. A list variable may be of any type. For example, the sequence

```
      READ (6,5) V
    5 FORMAT (A4)
```

causes four alphanumeric characters to be read from the card reader and stored in the variable V.

For a double precision variable the maximum number of characters transferred is ten (i.e., two storage locations are used); for all other variables, the maximum is five characters. If *w* exceeds the maximum, the leftmost characters are lost on input and are replaced with blanks on output. If, on input, *w* is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, *w* is less than the maximum, the leftmost *w* characters are transmitted to the external medium. Since for complex variables each word requires a separate field specification, the maximum value of *w* is 5. For example,

```
      COMPLEX C
      ACCEPT 1,C
    1 FORMAT (2A5)
```

transfers ten alphanumeric characters from the user's I/O terminal (TTY) into complex variable C.

13.2.8 Transferring Alphanumeric Data Directly Into or From FORMAT Statements

Alphanumeric data may be transmitted directly into or from the FORMAT statement by two different methods: H-conversion, or the ***use of single quotes (i.e., a literal field descriptor).***

In H-conversion, the alphanumeric string is specified in the form nH, where *H* is the control character and *n* is the total number of characters (including blanks) in the string. For example, the following statement sequence may be used to print the words PROGRAM COMPLETE on the device LPT:

```
      PRINT 101
  101 FORMAT (17HbPROGRAMbCOMPLETE)
```

Read and write operations of this type are initiated by I/O statements which reference a format statement and a logical device but do not contain an I/O list (see preceding example).

Write transfers from a FORMAT statement cause the contents of the statement field descriptor to be output to a specified logical device. The contents of the field descriptor, however, remain unchanged.

Read transfers with a FORMAT statement cause the contents of the field descriptors involved to be replaced by the characters input from the specified logical device.

Alphanumeric data is stored in a field descriptor right-justified. If the data input into a field has fewer characters than the field, leading blanks are added to fill the field. If the data input is larger than the field of the descriptor, the excess leftmost characters are lost.

Examples

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

cause the string PROGRAM COMPLETE to be output to the file on device 1.

Assuming the string START on device 1, the sequence

```
      READ (1,101)
101   FORMAT (17H␣PROGRAM␣COMPLETE)
```

would change the contents of statement 101 to

```
101 FORMAT (17HSTART␣␣␣␣␣␣␣␣␣␣␣␣␣)
```

The foregoing functions may also be accomplished by a *literal field descriptor consisting of the desired character string enclosed within apostrophes (i.e., 'string'). For example, the descriptors*

```
101 FORMAT (17H␣PROGRAM␣COMPLETE)
```

and

```
101 FORMAT ('␣PROGRAM␣COMPLETE')
```

may be used in the same manner.

The result of literal conversion is the same as H-conversion; on input, the characters between the apostrophes are replaced by input characters and, on output, the characters between the apostrophes (including blanks) are written as part of the output data.

An apostrophe character within a literal field should be represented by two successive apostrophe marks; otherwise, the statement containing the field will not compile. For example, the statement sequence

```
50 FORMAT ('DON''T')
   PRINT 50
```

will compile and will cause the word DON'T to be output on the line printer. The statement

```
50 FORMAT ('DON'T')
```

however, will cause a compile error.

13.2.9 Mixed Numeric and Alphanumeric Fields

An alphanumeric field descriptor may be placed among other fields of the format. For example, the statement:

```
FORMAT (15,7H␣FORCE=F10.5)
```

may be used to output the line:

```
␣␣22␣FORCE=␣␣17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown in the foregoing statement.

When a comma delimiter is omitted from a format specification, format control associates as much information as possible with the leftmost of the two field descriptors.

13.2.10 Multiple Record Specifications

To handle a group of input/output records where different records have different field descriptors, a slash is used to indicate a new record. For example, the statement

```
FORMAT (308/I5,2F8.4)
```

is equivalent to

```
FORMAT (308)
```

for the first record, and

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

for the second record.

Separating commas may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records will be written on output or skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written on output or n-1 records skipped on input.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that the transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one or level zero if no level one group exists.

Thus, the statement

```
FORMAT (F7.2,(2(E15.5E15.4),17))
level 0      level 1          level 1  level 0
```

causes the format

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

to be used on the first record.

As a further example, consider the statement

```
FORMAT (F7.2/(2(E15.5,E15.4),I7))
```

The first record has the format

```
F7.2
```

and successive records have the format

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

13.2.11 Record Formatting Field Descriptors

Two field descriptors, Tw and nX, may be used to position data within a record.

The field descriptor Tw may be used to specify the character position (external form) in which a record begins. In the Tw field descriptor the letter T is the control character and w is an unsigned integer constant which specifies the character position, in a FORTRAN-10 record, where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position since the first character of the output buffer is a forms control character and is not printed. It is recommended that the first field specification of the output format be 1X, except where a forms control character is used.

NOTE

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

Examples

The statement sequence

```
   PRINT 2
2  FORMAT (T50,'BLACK'T30,'WHITE')
```

causes the following line to be printed

```
      WHITE                BLACK
        ↑                    ↑
(print position 29)   (print position 49)
```

The statement sequence

```
1  FORMAT (T35,'MONTH')
   READ (2,1)
```

causes the first 34 characters of the input data associated with logical unit 2 to be skipped, and the next five characters to replace the characters M,O,N,T, and H in storage. If an input record containing

ABCbbbXYZ

is read with the format specification

10 FORMAT (T7,A3,T1,A3)

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

The field descriptor nX may be used to introduce blanks into output records or to skip characters of input records. The letter *X* specifies the operation and *n* is a positive integer that specifies the number of character positions to be either made blanks (output) or skipped (input).

Example

The statement

```
FORMAT (5Hβ STEP15,10X2HY=F7.3)
```

may be used to print the line

```
STEPβββ28βββββββββββY=b-3.872
```

13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS

The first character of an ASCII record may be used to control the spacing operations of the line printer or Teletype terminal printer unit on which the record is being printed. The control character desired is specified by beginning the FORMAT field specification for the ASCII record to be output with 1Ha. . . where *a* is the desired control character. The control characters permitted in FORTRAN-10 and the effect each has on the printing device are described in Table 13-5.

Table 13-5
FORTRAN-10 Print Control Characters

FORTRAN Character	Printer Character	Octal Value	Effect
space	LF	012	Skip to next line with form feed after 60 lines
0 zero	LF,LF	012	Skip a line
1 one	FF	014	Form feed – go to top of next page
+ plus			Suppress skipping – overprint the line
* asterisk	DC3	023	Skip to next line with no form feed
- minus	LF,LF,LF	012	Skip two lines
2 two	DLE	020	Space 1/2 of a page
3 three	VT	013	Space 1/3 of a page
/ slash	DC4	024	Space 1/6 of a page
. period	DC2	022	Triple space with a form feed after every 20 lines printed
, comma	DC1	021	Double space with a form feed after every 30 lines printed

Note: Printer control characters DLE, DC1, DC2, DC3, and DC4 affect only the line printer.

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 14
DEVICE CONTROL STATEMENTS

14.1 INTRODUCTION

The following device control statements may be used in FORTRAN-10 source programs:

1. REWIND
2. ***UNLOAD***
3. BACKSPACE[1]
4. ENDFILE
5. ***SKIPRECORD***[1]
6. ***SKIPFILE,*** and
7. ***BACKFILE***

The general form of the foregoing device control statements is

keyword u
keyword (u)

where

keyword	is the statement name
u	is the FORTRAN-10 logical device number (Chapter 10, Table 10-1)

The operations performed by the device control statement are normally used only for magnetic tape device (MTA). In FORTRAN-10, however, the device control operations are simulated for disk devices.

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

14.2 REWIND STATEMENT

Descriptions of the form and use of the REWIND statement follow:

Form: REWIND u

Use: Move the file contained by device *u* to its initial (load) point. If the medium is already at its load point, this statement has no effect. Subsequent READ or WRITE statements that reference device *u* will transfer data to or from the first record located on the medium mounted on device *u.*

Example: REWIND 16

14.3 UNLOAD STATEMENT

Descriptions of the form and use of the UNLOAD statement follow:

Form: UNLOAD u

Use: Move the medium contained on device u past its load point until it has been completely rewound onto the source reel.

Example: UNLOAD 16

14.4 BACKSPACE STATEMENT

Descriptions of the form and use of the BACKSPACE statement follow:

Form: BACKSPACE u

Use: Move the medium contained on device *u* to the start of the record that precedes the current record. If the preceding record prior to execution of this statement was an endfile record, the endfile record becomes the next record after execution. If the current record is the first record of the file, this statement has no effect.

NOTE

This statement cannot be used for files set up for random access or NAMELIST-controlled I/O operations.

Example: BACKSPACE 16

14.5 END FILE STATEMENT

Descriptions of the form and use of the END FILE statement follow:

Form: END FILE u

Use: Write an endfile record in the file located on device *u.* The endfile record defines the end of the file which contains it. If an endfile record is reached during an I/O operation initiated by a statement that does not contain an END= option, the operation of the current program is terminated.

Example: END FILE 16

14.6 SKIP RECORD STATEMENT

Descriptions of the form and use of the SKIP RECORD statement follow:

Form: *SKIP RECORD u*

Use: *In accessing the file located on device u, skip the record immediately following the current (last accessed) record. The repeat option may be used to cause any desired number of records to be skipped.*

Example: *SKIP RECORD 16*

14.7 SKIP FILE STATEMENT

Descriptions of the form and use of the SKIP FILE statement follow:

Form: *SKIP FILE u*

Use: *In accessing the medium located on unit u, skip the file immediately following the current (last accessed) file. If the number of SKIP FILE operations specified exceeds the number of following files available, an error will occur.*

Example: *SKIP FILE 01*

14.8 BACKFILE STATEMENT

Descriptions of the form and use of the BACKFILE statement follow:

Form: *BACKFILE u*

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

If the number of BACKFILE operations performed exceeds the number of preceding files, completion of the last operation will move the medium to the start of the first file on the medium.

Example: *BACKFILE 20*

14.9 SUMMARY OF DEVICE CONTROL STATEMENTS

The form and use of the FORTRAN-10 device control statements are summarized in Table 14-1.

Table 14-1
Summary of FORTRAN-10 Device Control Statements

Statement Form	Use
REWIND *u*	Rewind medium to its load point
UNLOAD u	*Rewind medium onto its source reel*
END FILE *u*	Write an endfile record in to the current file
SKIP RECORD u	*Skip the next record*
SKIP FILE u	*Skip the next file*
BACKFILE u	*Move medium backwards 1 file*
BACKSPACE *u*	Move medium back one record

CHAPTER 15
SUBPROGRAM STATEMENTS

15.1 INTRODUCTION

Procedures that are used repeatedly by a program may be written once and then referenced each time the procedure is required. Procedures that may be referenced are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that may be compiled separately). The kinds of FORTRAN-10 procedures that may be referenced are:

a. statement functions

b. intrinsic functions

c. external functions, and

d. subroutines

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

15.1.1 Dummy and Actual Arguments

Since subprograms may be referenced at more than one point throughout a program, many of the values used by the subprogram may be changed each time it is used. Dummy arguments in subprograms represent the actual values to be used which are passed to the subprogram when it is called.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments which they represent and whether the actual arguments are variables, array elements, arrays, subroutine names or the names of external functions. Each dummy argument must be used within a function or subroutine as if it were a variable, array, array element, subroutine, or external function identifier. Dummy arguments are given in an argument list associated with the identifier assigned to the subprogram; actual arguments are normally given in an argument list associated with a call made to the desired subprogram. (Examples of argument lists are given in the following paragraphs.)

The position, number, and type of each dummy argument in a subprogram list must agree with the position, number, and type of each actual argument given in the argument list of the subprogram reference.

Dummy arguments may be

a. variables

b. array names

c. subroutine identifiers

d. function identifiers, or

*e. statement label identifiers which are denoted by the symbol *, $, or &.*

When a subprogram is referenced, its dummy arguments are replaced by the corresponding actual arguments supplied in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the results of the subprogram computations will be unpredictable. Argument association may be carried through more than one level of subprogram reference if a valid association is maintained through each level. The dummy/actual argument associations established when a subprogram is referenced are terminated when the desired subprogram operations are completed.

The following rules govern the use and form of dummy arguments:

a. The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time the procedure is referenced.

b. Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.

c. A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding actual argument.

d. An array dummy argument should have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. Each element of a dummy array is associated directly with the corresponding elements of the actual array.

e. A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.

f. A dummy argument representing an external function must have an external function as its actual argument.

g. A dummy argument may be defined or redefined in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then elements of the array may be redefined.

Additional information regarding the use of dummy and actual arguments is given in the description of how subprograms are defined and referenced.

15.2 STATEMENT FUNCTIONS

Statement functions define an internal subprogram in a single statement. The general form of a statement function is:

name (arg1,arg2,. . .,argn)=E

where

name	is a user-formulated name comprised of from 1 to 6 characters. The name used must conform to the rules for symbolic names given in Paragraph 3.3. The type of a statement function is determined either by the first character of its name or by being declared in an explicit or implicit type statement.
(arg1. . .argn)	represents a list of dummy arguments.
E	is an arbitrary expression.

The expression *E* of a statement function may be any legitimate arithmetic expression which may use the given dummy arguments and indicates how they are combined to obtain the desired value. The expression may reference intrinsic functions (Paragraph 15.3) or any other defined statement function, or call an external function. It may not reference any function that directly or indirectly references the given statement function or any subprogram in the chain of references which lead to this function.

Statement functions produce only one value, the result of the expression which it contains. A statement function cannot reference itself.

All statement functions within a program unit must be defined before the first executable statement of the program unit. When used, the statement function name must be followed by an actual argument list enclosed within parentheses and may appear in any arithmetic or logical expression.

Examples

```
SSQR(K)=(K*(K+1)*2*K+1)/6
ACOSH(X)=(EXP(X/A)+EXP(-X/A))/2.0
```

15.3 INTRINSIC FUNCTIONS

Intrinsic functions are subprograms that are defined and supplied by FORTRAN-10. An intrinsic function is referenced by using its assigned name as an operand in an arithmetic or logical expression. The names of the FORTRAN-10 intrinsic functions, the type of the arguments which each accepts, and the function it performs are described in Table 15-1. These names always refer to the intrinsic function unless declared in an EXTERNAL statement or conflicting explicit type statement.

Table 15-1
Intrinsic Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Absolute value:					
Real	ABS*	arg	1	Real	Real
Integer	IABS	arg	1	Integer	Integer
Double precision	DABS	arg	1	Double	Double
Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real
Conversion:					
Integer to real	FLOAT*		1	Integer	Real
Real to integer	IFIX*	Sign of arg * largest integer $\leq \lvert arg \rvert$	1	Real	Integer
Double to real	SNGL		1	Double	Real
Real to double	DBLE		1	Real	Double
Integer to double	DFLOAT		1	Integer	Double
Complex to real (obtain real part)	REAL		1	Complex	Real
Complex to real (obtain imaginary part)	AIMAG		1	Complex	Real
Real to complex	CMPLX	$c=Arg_1+i*Arg_2$	2	Real	Complex
Truncation:					
Real to real	AINT	Sign of arg * largest integer $\leq \lvert arg \rvert$	1	Real	Real
Real to integer	INT*		1	Real	Integer
Double to integer	IDINT		1	Double	Integer
Remaindering:					
Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real
Integer	MOD		2	Integer	Integer
Double precision	DMOD		2	Double	Double
Maximum value:					
	AMAX0	$Max(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real
	AMAX1*		≥ 2	Real	Real
	MAX0		≥ 2	Integer	Integer
	MAX1		≥ 2	Real	Integer
	DMAX1		≥ 2	Double	Double
Minimum Value:					
	AMIN0	$Min(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real
	AMIN1		≥ 2	Real	Real
	MIN0		≥ 2	Integer	Integer
	MIN1		≥ 2	Real	Integer
	DMIN1		≥ 2	Double	Double

*In line functions.

Table 15-1 (Cont)
Intrinsic Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Transfer of Sign:					
Real	SIGN*	$\left\{ Sgn(Arg_2)*\lvert Arg_1\rvert \right\}$	2	Real	Real
Integer	ISIGN		2	Integer	Integer
Double precision	DSIGN		2	Double	Double
Positive Difference:					
Real	DIM	$\left\{ Arg_1 - Min(Arg_1, Arg_2) \right\}$	2	Real	Real
Integer	IDIM		2	Integer	Integer

*In line functions.

15.4 EXTERNAL FUNCTIONS

External functions are function subprograms that consist of a FUNCTION statement followed by a sequence of FORTRAN-10 statements that define one or more desired operations; subprograms of this type may contain one or more RETURN statements and must be terminated by an END statement. Function subprograms are independent programs that may be referenced by other programs.

The FUNCTION statement that identifies an external function has the form

type FUNCTION name (arg1,arg2,. . .,argn)

where

type is an optional type specification of the form INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL.

name is the name assigned to the function. The name may consist of from 1 to 6 characters, the first of which must be alphabetic.

(arg1,. . .,argn) is a list of dummy arguments.

If *type* is not given in the FUNCTION statement, the type of the function may be assigned, by default, according to the first character of its name, or may be specified by an IMPLICIT statement or by an explicit statement given within the subprogram itself.

If the user wants to use the same name for a user-generated function as the name of a library basic external function, the desired name must be declared in an EXTERNAL statement.

The following rules govern the structuring of a FUNCTION subprogram:

a. The symbolic name assigned a FUNCTION subprogram must also be used as a variable name in the subprogram. During each execution of the subprogram this variable must be defined and, once defined, may be referenced as redefined. The value of the variable at the time of execution on any RETURN statement is the value of the subprogram.

NOTE

A RETURN statement returns control to the calling statement that initiated the execution of the subprogram. See Paragraph 15.4.1 for a description of this statement.

b. The symbolic name of a FUNCTION subprogram must not be used in any nonexecutable statement in the subprogram except in the initial FUNCTION statement.

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

d. The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

e. The function subprogram may contain any FORTRAN-10 statement except BLOCK DATA, SUBROUTINE PROGRAM, another FUNCTION statement or any statement that directly or indirectly references the function being defined or any subprogram in the chain of subprograms leading to this function.

f. The function subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statement signifies a logical conclusion of the computation made by the subprogram and returns the computed function value and control to the calling program. A subprogram may have more than one RETURN statement.

The END statement specifies the physical end of the subprogram and implies a return.

15.4.1 Basic External Functions

FORTRAN-10 contains a group of predefined external functions which are referred to as basic functions. Table 15-2 describes each basic function, its name, and its use. These names always refer to the basic external functions unless declared in an EXTERNAL or conflicting explicit type statement.

15.4.2 Generic Function Names

The compiler will generate a call to the proper FORTRAN-10 supplied function, depending on the type of the arguments, for the following generic function names.

ABS
AMAX1
AMIN1
ATAN
ATAN2
COS
INT
MOD
SIGN
SIN
SQRT
EXP
ALOG
ALOG10

A name loses its generic properties if it appears in an EXTERNAL or explicit type statement.

15.5 SUBROUTINE SUBPROGRAMS

A subroutine is an external computational procedure which is identified by a SUBROUTINE statement and may or may not return values to the calling program. The SUBROUTINE statement used to identify a subprogram of this type has the form:

SUBROUTINE *name*(arg1,arg2,. . .,argn)

where

name	is the symbolic name of the subroutine to be defined.
(arg1,. . .,argn)	is an optional list of dummy arguments.

Table 15-2
Basic External Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function
Exponential:					
Real	EXP	$\left\{ {}_{e}\text{Arg} \right\}$	1	Real	Real
Double	DEXP		1	Double	Double
Complex	CEXP		1	Complex	Complex
Logarithm:					
Real	ALOG	$\log_e(\text{Arg})$	1	Real	Real
	ALOG10	$\log_{10}(\text{Arg})$	1	Real	Real
Double	DLOG	$\log_e(\text{Arg})$	1	Double	Double
	DLOG10	$\log_{10}(\text{Arg})$	1	Double	Double
Complex	CLOG	$\log_e(\text{Arg})$	1	Complex	Complex
Square Root:					
Real	SQRT*	$(\text{Arg})^{1/2}$	1	Real	Real
Double	DSQRT	$(\text{Arg})^{1/2}$	1	Double	Double
Complex	CSQRT	$(\text{Arg})^{1/2}$	1	Complex	Complex
Sine:					
Real (radians)	SIN*	$\left\{ \sin(\text{Arg}) \right\}$	1	Real	Real
Real (degrees)	SIND		1	Real	Real
Double (radians)	DSIN		1	Double	Double
Complex	CSIN		1	Complex	Complex
Cosine:					
Real (radians)	COS*	$\left\{ \cos(\text{Arg}) \right\}$	1	Real	Real
Real (degrees)	COSD		1	Real	Real
Double (radians)	DCOS		1	Double	Double
Complex	CCOS		1	Complex	Complex
Hyperbolic:					
Sine	SINH	sinh(Arg)	1	Real	Real
Cosine	COSH	cosh(Arg)	1	Real	Real
Tangent	TANH	tanh(Arg)	1	Real	Real
Arc sine	ASIN	asin(Arg)	1	Real	Real
Arc cosine	ACOS	acos(Arg)	1	Real	Real
Arc tangent					
Real	ATAN*	atan(Arg)	1	Real	Real
Double	DATAN	datan(Arg)	1	Double	Double
Two REAL arguments	ATAN2*	$\text{atan}(\text{Arg}_1/\text{Arg}_2)$	2	Real	Real
Two DOUBLE arguments	DATAN2	$\text{atan}(\text{Arg}_1/\text{Arg}_2)$	2	Double	Double

*Generates in-line code.

Table 15-2 (Cont)
Basic External Functions

Function	Mnemonic	Definition	Number of Arguments	Type of	
				Argument	Function
Complex Conjugate	CONJG	Arg=X+iY,CONJG=X-iY	1	Complex	Complex
Random Number	RAN	Result is a random number in the range of 0 to 1.0.	1 Dummy Argument	Integer, Real, Double, or Complex	Real

*Generates in-line code.

The following rules control the structuring of a subroutine subprogram:

a. The symbolic name of the subprogram must not appear in any statement within the defined subprogram except the SUBROUTINE statement itself.

b. The given dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement within the subprogram.

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

d. The subroutine subprogram may contain any FORTRAN-10 statement except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that either directly or indirectly references the subroutine being defined or any of the subprograms in the chain of subprogram references leading to this subroutine.

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

f. The subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statements indicate the logical end of a computational routine; the END statement signifies the physical end of the subroutine.

g. Subroutine subprograms may have as many entry points as desired (see description of ENTRY statement given in Paragraph 15.4.1).

15.5.1 Referencing Subroutines (CALL Statement)

Subroutine subprograms must be referenced using a CALL statement of the following form:

```
CALL name(arg1,arg2,. . .,argn)
```

where

name is the symbolic name of the desired subroutine subprogram.

(arg1,. . .,argn) is an optional list of actual arguments. If the list is included, the given actual arguments must agree in order, number, and type with the corresponding dummy arguments given in the defining SUBROUTINE statement.

The use of literal constants is an exception to the rule requiring agreement of type between dummy and actual arguments. An actual argument in a CALL statement may be:

a. a constant

b. a variable name

c. an array element identifier

d. an array name

e. an expression

f. the name of an external subroutine, or

g. a statement label.

Example

The subroutine

```
SUBROUTINE MATRIX(I,J,K,M,*)
.
.
.
END
```

may be referenced by

```
CALL MATRIX(10,20,30,40,$101)
```

15.5.2 FORTRAN-10 Supplied Subroutines

FORTRAN-10 provides the user with an extensive group of predefined subroutines. The descriptions and names of these predefined subroutines are given in Table 15-3.

15.6 RETURN STATEMENT AND MULTIPLE RETURNS

The RETURN statement causes control to be returned from a subprogram to the calling program unit. This statement has the form

```
RETURN    (standard return)
```

or

```
RETURN e    (multiple returns)
```

where e represents an integer constant, variable, or expression. The execution of this statement in the first of the foregoing forms (i.e., standard return) causes control to be returned to the statement of the calling program which follows the statement that called the subprogram.

The multiple returns form of this statement (i.e., RETURN e) enables the user to select any labeled statement of the calling program as a return point. When the multiple returns form of this statement is executed, the assigned or calculated value of e specifies that the return is to be made to the eth statement label in the argument list of the calling statement. The value of e should be a positive integer which is equal to or less than the number of statement labels given in the argument list of the calling statement. If e is less than 1 or is larger than the number of available statement labels, a standard return operation is performed.

NOTE
*A dummy argument for a statement label must be either a *, $, or & symbol.*

Any number of RETURN (standard return) statements may be used in any subprogram. The use of the multiple returns form of the RETURN statement, however, is restricted to SUBROUTINE subprograms. The execution of a RETURN statement in a main program will terminate the program.

Example

Assume the following statement sequence in a main program:

```
      .
      .
      .
      CALL EXAMP(I,$10,K,$20,M,$30)
      GO TO 101
      .
      .
      .
10    . . . . . .
      .
      .
      .
15    . . . . . .
      .
      .
      .
20    . . . . . .
      .
      .
      .
```

Assume the following statement sequence in the called SUBROUTINE subprogram:

```
SUBROUTINE EXAMP (L,*,M,*,N,*)
.
.
RETURN
.
.
RETURN
.
.
RETURN(C/D)
.
.
END
```

Each occurrence of RETURN returns control to the statement GO TO 101 in the calling program.

If, on the execution of the RETURN(C/D) statement, the value of (C/D) is:

Less than or equal to:	*The following is performed:*
0	*a standard return to the GO TO 101 statement is made*
1	*the return is made to statement 10*
2	*the return is made to statement 15*
3	*the return is made to statement 20*

Greater than or equal to:	*The following is performed:*
4	*a standard return to the GO TO 101 statement is made.*

15.6.1 Referencing External FUNCTION Subprograms

An external function subprogram is referenced by using its assigned name as an operand in an arithmetic or logical expression in the calling program unit. The name must be followed by an actual argument list. The actual arguments in an external function reference may be:

a. a variable name

b. an array element identifier

c. an array name

d. an expression

e. a statement number

f. the name of another external procedure (FUNCTION or SUBROUTINE).

NOTE

Any subprogram name to be used as an argument to another subprogram must first appear in an EXTERNAL statement (Chapter 6) in the calling program unit.

Example

The subprogram defined as:

```
INTEGER FUNCTION ICALC(X,Y,Z)
.
.
.
RETURN
END
```

may be referenced in the following manner:

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

15.7 MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT)

FORTRAN-10 provides an ENTRY statement which enables the user to specify additional entry points into an external subprogram. This statement used in conjunction with a RETURN statement enables the user to employ only one computational routine of a subprogram which contains several such routines. The form of the ENTRY statement is:

ENTRY name(arg1,arg2,. . .,argn)

where

name — *is the symbolic name to be assigned the desired entry point*

(arg1,. . .,argn) — *is an optional list of dummy arguments. This list may contain*

a. variable names

b. array declarators

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

*d. an address constant denoted by either a *, $, or & symbol*

The rules for the use of an ENTRY statment follow.

a. The ENTRY statement allows entry into a subprogram at a place other than that defined by the subroutine or function statement. Any number of ENTRY statements may be included in an external subprogram.

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

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

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

e. An ENTRY statement may not appear in a main program, nor may a subprogram reference itself through its entry points.

f. An ENTRY statement may not appear in the range of a DO or an extended DO statement construction.

g. The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in SUBROUTINE or FUNCTION statements or any other ENTRY statement in the subprogram. However, the arguments for each call or function reference must agree with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that is referenced.

h. Entry into a subprogram initializes the dummy arguments of the referenced ENTRY statement, all appearances of these arguments in the entire subprogram are initialized.

i. A dummy argument may not be referenced unless it appears in the dummy list of an ENTRY, SUBROUTINE, or FUNCTION statement by which the subprogram is entered.

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

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

l. The value of the function must be returned by using the current entry name.

Table 15-3
FORTRAN-10 Library Subroutines

Subroutine Name	Effect
AXIS	CALL AXIS(X,Y,ASC,NASC,S,THETA,XMIN,DX) Causes an axis with tic marks and scale values at 1-inch increments to be drawn. An identifying label may also be plotted along the axis. Parameters X and Y specify the start of the axis. The axis is plotted, starting at X, Y, at an angle of THETA degrees for a distance of S inches. The angle THETA is usually either 0 (X axis) or 90.0 (Y axis). Characters NASC of array ASC are plotted as a label for the axis drawn. If NASC is positive, the tic marks, label, and scale values are placed on the positive (clockwise) side of the axis; if NASC is negative, the foregoing items are placed on the negative (counterclockwise) side of the axis. Parameter XMIN is the value of the scale at the beginning of the axis; parameter DX is the change in scale for a 1-inch increment. The values of XMIN and DX may be determined by subroutine SCALE.
DATE	Places today's date as left-justified ASCII characters into a dimensioned 2-word array. CALL DATA (array) where array is the 2-word array. The date is in the form dd-mmm-yy where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-digit month (e.g., MAR), and yy is a 2-digit year. The date is stored in ASCII code, left-justified, in the two words.
DEFINE FILE	A DEFINE FILE call can be used to establish and define the structure of each file to be used for random access I/O operations. **NOTE** **The OPEN statement may be used to perform the same functions as DEFINE FILE.** The format of a DEFINE FILE call may be CALL DEFINE FILE (u,s,v,f,pj,pg) where

Table 15-3 (Cont)
FORTRAN-10 Library Subroutines

Subroutine Name	Effect
DEFINE FILE (cont)	*u* = logical FORTRAN-10 device numbers. *s* = the size of the records which comprise the file being defined. The argument *s* may be an integer constant or variable. *v* = an associated variable. The associated variable is an integer variable that is set to a value that points to the record that immediately follows the last record transferred. This variable is used by the FIND statement (Chapter 10). At the end of each FIND operation the variable is set to a value that points to the record found. The variable *v* cannot appear in the I/O list of any I/O statement that accesses the file set up by the DEFINE FILE statement. *f* = filename to be given the file being defined. *pj* = user's project number. *pg* = user's programmer's number. NOTE Numbers pj and pg identify the User's File Directory. Example The statement DEFINE FILE 01(1000,10,FORTFL.DAT,ASCVAR) establishes a file named FORTFL.DAT on device 01 (i.e., disk) which contains 1000 words divided into 100 10-word records.
DUMP	Causes particular portions of core to be dumped and is referred to in the following form: CALL DUMP (L_1,U_1,F_1,. . .,L_n,U_n,F_n) where L_1 and U_1 are the variable names which give the limits of core memory to be dumped. Either L_1 or U_1 may be upper or lower limits. F_1 is a number indicating the format in which the dump is to be performed: 0 = octal, 1 = real, 2 = integer, and 3 = ASCII. If F is not 0, 1, 2, 3, the dump is in octal. If F_n is missing, the last section is dumped in octal. If U_n and F_n are missing, an octal dump is made from L to the end of the job area. If L_n, U_n, and F_n are missing, the entire job area is dumped in octal. The dump is terminated by a call to EXIT.

Table 15-3 (Cont)
FORTRAN-10 Library Subroutines

Subroutine Name	Effect
ERRSET	Allows the user to control the typeout of execution-time arithmetic error messages, ERRSET is called with one argument in integer mode. CALL ERRSET(N) Typeout of each type of error message is suppressed after N occurrences of that error message. If ERRSET is not called, the default value of N is 2.
EXIT	Returns control to the Monitor and, therefore, terminates the execution of the program.
ILL	Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating point/double precision input, the corresponding word is set to zero. CALL ILL
LEGAL	Clears the ILLEG flag. If the flag is set and an illegal character is encountered in the floating point/double precision input, the corresponding word is set to zero. CALL LEGAL
LINE	Causes a line to be drawn through the N points specified by (X(1),Y(1)),(X(2),Y(2)). . .(X(N),Y(N)) where the elements of X and Y are spaced K words apart in storage. CALL LINE (X,Y,N,K)
MKTBL	CALL MKTBL(I,J) Specifies a special character set where I is the number to be assigned the set and J contains the starting address of a character table of 200_8 consecutive words. In each character table word the left half contains the number of strokes in the character (0 if nothing is to be plotted for the word) and the right half contains the address of the table of strokes for the character.
NUMBER	CALL NUMBER(X,Y,SIZE,FNUM,THETA,NDIGIT) Causes a floating point number to be plotted as text. Parameters X, Y, SIZE and THETA have the same meanings as for the SYMBOL call. Parameter NDIGIT is the number of digits plotted to the right of the decimal point. If NDIGIT is a negative value, only the integer part of the number is plotted. FNUM specifies the number to be plotted.
PDUMP	CALL PDUMP(L_1,U_1,F_1,. . .,L_n,U_n,F_n) The arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.

Table 15-3 (Cont)
FORTRAN-10 Library Subroutines

Subroutine Name	Effect
PLOT	CALL PLOT(X,Y,IPEN) Move the pen in a straight line from its current position to the position specified by X,Y. If IPEN=3, the pen is raised before the movement; if IPEN=2 the pen is lowered before movement; if IPEN=1 the pen is left unchanged from its previous state. If the value of IPEN is negative (-1, -2 or -3) the pen action is the same as for the corresponding positive values except that on completion of the indicated motion the new pen position is taken as a new origin and the output buffer is sent to the plotter. The plotter is not released on completion of the specified movement.
PLOTS	CALL PLOTS (I) The plotter setup routine is called. If the plotter is not available, I is set to -1; if it is available, I is set to 0. This call must be the first plotter routine called.
RELEAS	CALL RELEAS(unit*) Closes out I/O on a device initialized by the FORTRAN Operating System and returns it to the uninitialized state.
SAVRAN	SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SCALE	CALL SCALE(X,N,X,XMIN,DX) Selects scale values for an AXIS call where X and N specify a 1-dimensional array X with the length N. Parameter S specifies the length of the desired axis, SCALE determines a value of DX which allows X to be plotted in S inches. XMIN is selected as the smallest element of the array X, and is truncated to be a multiple of DX.
SETABL	CALL SETABL(I,J) Specifies a character set where I is an integer which gives the number of the desired character set. If a character set has been defined by I, the value of J is set to 0; if not, J is set to -1. The standard ASCII character set is defined as 1.
SETRAN	SETRAN has one argument which must be a non-negative integer $< 2^{31}$. The starting value of the function RAN is set to one value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.
SLITE(i)	Turn sense lights on or off. i is an integer expression. For $1 \leqslant i \leqslant 36$ sense light i will be turned on. If i=0, all sense lights will be turned off.

Table 15-3 (Cont)
FORTRAN-10 Library Subroutines

Subroutine Name	Effect
SLITET(i,j)	Checks the status of sense light i and sets the variable j accordingly and turns off sense light i. If i is on, j is set to 1; and if i is off, j is set to 2.
SSWTCH(i,j)	Checks the status of data switch i ($0 \leq i \leq 35$) and sets the variable j accordingly. If i is set OFF, j is set to 1; and, if i is ON, j is set to 2.
SYMBOL	CALL SYMBOL(X,Y,SIZE,BCD,THETA,NBCD) Raise pen and move it to position specified by X and Y. Lower pen and plot characters found in array ASC. Parameter SIZE specifies the height of the characters plotted in inches (floating point values); THETA specifies the direction of the base line on which the text of array ASC is to be plotted, and NASC specifies the number of characters in array ASC.
TIME	Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh:mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form ss.t where ss is the seconds and t is the tenths of a second.
WHERE	CALL WHERE(X,Y) Variables X and Y are set to the values which identify the current position of the pen.

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 16
BLOCK DATA SUBPROGRAMS

16.1 INTRODUCTION

Block data subprograms are used to initialize data to be stored in any common areas. Only specification and DATA statements are permitted (i.e., DATA, COMMON, DIMENSION, EQUIVALENCE, and TYPE) in block subprograms. A subprogram of this type must start with a BLOCK DATA statement.

If any entry of a labeled common block is initialized by a BLOCK DATA subprogram, the entire block must be included even though some of the elements of the block do not appear in DATA statements.

Initial values may be entered into more than one labeled common block in a single subprogram of this type.

An executable program may contain more than one block data subprogram.

16.2 BLOCK DATA STATEMENT

The form of the BLOCK DATA statement is

BLOCK DATA name

where

name is a symbolic name given to identify the subprogram.

APPENDIX A
ASCII-1968 CHARACTER CODE SET

The character code set defined in the X3.4-1968 Version of the American National Standard for Information Interchange (ASCII) is given in the following matrix.

1st 2 octal digits	Last octal digit 0	1	2	3	4	5	6	7
00x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	DEL
01x	BS	HT	LF	VT	FF	CR	SO	SI
02x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
03x	CAN	EM	SUB	ESC	FS	GS	RS	US
04x	♮	!	"	#	$	%	&	'
05x	(	)	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	\	]	^ (↑)	_ (←)
14x		a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	{	|	}	(ESC)	DEL

Characters inside parentheses are ASCII-1963 Standard.

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

APPENDIX B
WRITING USER PROGRAMS

The DECsystem-10 FORTRAN-10 compiler is distributed under the name FORTRAN. The FORTRAN compiler is a highly optimizing compiler particularly useful in a scientific production environment. This appendix contains information about the use of the FORTRAN-10 compiler. It assumes the reader is familiar with the FORTRAN-10 language and the DECsystem-10 TOPS-10 monitor.

B.1 RUNNING THE FORTRAN-10 COMPILER

The command to run the FORTRAN-10 compiler is:

```
.R FORTRA
```

The compiler responds with an asterisk (*) and is then ready to accept a command line. A command is of the general form

```
object file name, listing file name = source file name(s).
```

The file names can be fully specified SFD paths. If no object file name is specified, no relocatable binary file is generated. If no listing file is specified, no listing is generated. Several source files may be specified; each must contain one or more complete FORTRAN-10 compilation units (i.e., a program or subprogram that terminates with an END line). The source files will be compiled in the order specified and stored in relocatable binary form in the single specified object file.

B.1.1 Switches Available with the FORTRAN-10 Compiler

Switches to the FORTRAN-10 compiler are accepted anywhere in the command line. They are totally position- and file-independent. The switches are shown in Table B-1.

Table B-1
FORTRAN-10 Compiler Switches

Switch	Minimum Abbreviation	Meaning
CROSSREF	C	Generate a file that can be input to the CREF program.
EXPAND	E	Include the octal-formatted version of the object file in the listing.
INCLUDE	I	Compile a D in card column 1 as a space.
KA10	KA	Compile code to run on a KA10 processor.
KI10	KI	Compile code to run on a KI10 processor.
MACROCODE	M	Add the mnemonic translation of the object code to the listing file.
NOERRORS	NOE	Do not print error messages on the terminal.
NOWARNINGS	NOW	Do not print warning messages on the terminal.
OPTIMIZE	O	Perform global optimization.
SYNTAX	S	Perform syntax check only.

Example:

```
.R FORTRAN
OFILE, LFILE = SFILE/M, S2FILE
```

The 1M switch will cause the macro code equivalent for both input files (SFILE and S2FILE) to appear on the listing.

If neither the /KA10 or /KI10 switch is used, code will be compiled for the processor type on which the compilation is occurring. The processor type of the code in the object file is indicated at the top of each listing page.

B.1.2 Monitor Commands

When both FORTRAN-10 and F40 are present in a DECsystem-10 system, users can specify which compiler is to be used. The switches /F10 or /F40 may be added to the commands

```
COMPILE
LOAD
EXECUTE
DEBUG
```

Example:

```
EXECUTE ROTOR /F10
```

If the switch is not specified and the file extension is FOR, the FORTRAN-10 compiler will be used. If the switch is not specified and the file extension is F4, the F40 compiler will be used.

B.2 READING A FORTRAN-10 LISTING

When a listing is requested from the FORTRAN-10 compiler, the listing contains the following information:

1. A printout of the source program text plus an internal sequence number assigned to each line by the compiler. This internal sequence number is referenced in any error or warning messages generated during the compilation.

2. A summary of the names and relative program locations (in octal) of scalars and arrays in the source program plus compiler generated variables.

3. All COMMON blocks and the relative locations (in octal) of the variables in each COMMON block.

4. A listing of the subprograms referenced (both user defined and FORTRAN-10 supplied library functions).

5. A summary of temporary locations generated by the compiler.

6. A heading on each page of the listing containing the program name (MAIN., subroutine or function principle entry) and the date and time of the compilation. Whether or not a specific processor switch (/KA10, /KI10) was used, the processor for which the code was generated is also at the top of the listing page. If the /OPTIMIZE switch was used, /OPT also appears on the listing page heading.

7. If the /MACRO switch was used, a mnemonic printout of the generated code (in a format similar to MACRO-10) is appended to the listing. This section of the listing has four fields:

   LINE: This column contains the internal sequence number of the line corresponding to the mnemonic code. It appears on the first instruction of the code sequence associated with that internal sequence number. An asterisk (*) indicates a compiler-inserted line.

   LOC: The relative location in the object program of the instruction.

   LABEL: Any program- or compiler-generated label. Program labels have the letter P appended. Compiler-generated labels are followed by the letter M.

   GENERATED CODE: The MACRO-10 mnemonic code.

8. A summary of all argument blocks generated by the compiler.

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

Name of program Name of source file The optimizer was selected

```
MAIN.    TIM2.FOR          FORTRAN V.1 /KI/OPT      9-NOV-73          11:18    PAGE 1
```

Each line is assigned an internal sequence number code was compiled for a KI processor

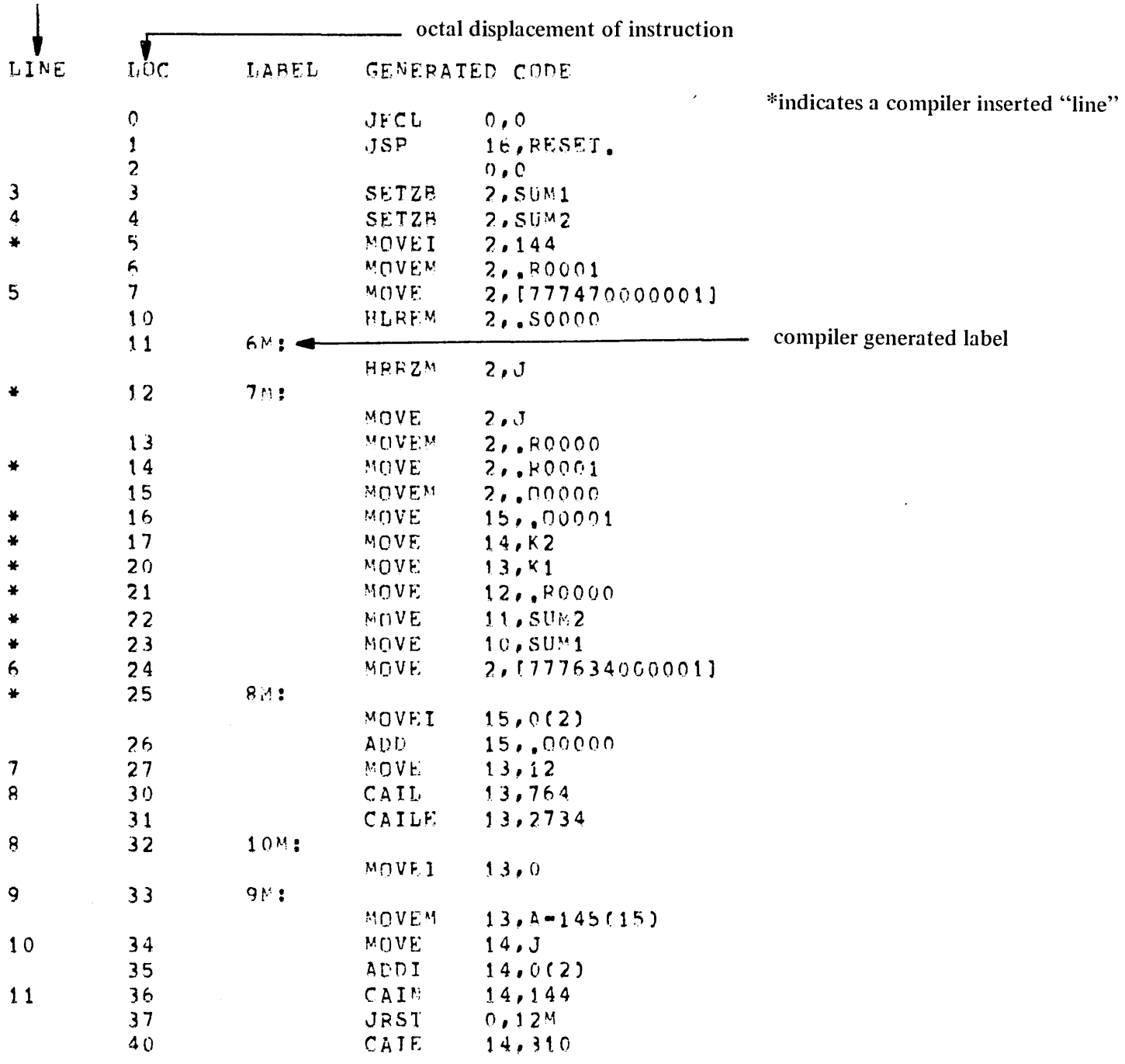
```
00001              IMPLICIT INTEGER (A-Z)
00002              DIMENSION A(100,200),B(100,200)
00003              SUM1=0
00004              SUM2=0
00005              DO 100 J=1,200
00006              DO 100 I=1,100
00007              K1=I*J
00008              IF (K1 .LT. 500 .OR. K1 .GT. 1500) K1=0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


SUBPROGRAMS CALLED


SCALARS AND ARRAYS          The relative address of all variables is given

K1       1        B        2        .R0001   47042    .R0000   47043    J        47044    A        47045
.S0001   116105   .S0000   116106   SUM2     116107   I        116110   .00001   116111   .00000   116112
K2       116113   SUM1     116114

TEMPORARIES
```


octal displacement of instruction
LINE
LOC
LABEL
GENERATED CODE
*indicates a compiler inserted "line"
compiler generated label
0 JFCL 0,0
1 JSP 16,RESET.
2 0,0
3 3 SETZB 2,SUM1
4 4 SETZB 2,SUM2
* 5 MOVEI 2,144
6 MOVEM 2,.R0001
5 7 MOVE 2,[777470000001]
10 HLREM 2,.S0000
11 6M:
HRRZM 2,J
* 12 7M:
MOVE 2,J
13 MOVEM 2,.R0000
* 14 MOVE 2,.R0001
15 MOVEM 2,.00000
* 16 MOVE 15,.00001
* 17 MOVE 14,K2
* 20 MOVE 13,K1
* 21 MOVE 12,.R0000
* 22 MOVE 11,SUM2
* 23 MOVE 10,SUM1
6 24 MOVE 2,[777634000001]
* 25 8M:
MOVEI 15,0(2)
26 ADD 15,.00000
7 27 MOVE 13,12
8 30 CAIL 13,764
31 CAILE 13,2734
8 32 10M:
MOVEI 13,0
9 33 9M:
MOVEM 13,A-145(15)
10 34 MOVE 14,J
35 ADDI 14,0(2)
11 36 CAIN 14,144
37 JRST 0,12M
40 CAIE 14,310

```
        41      13M:
                        CAIN    14,454
11      42      12M:
                        ADDI    14,1
12      43      11M:
                        MOVEM   14,B-145(15)
13      44              ADD     10,13
14      45              ADD     11,14
*       46      100P:  <-------------------------  program label
                1M:
                        ADD     12,J
        47              AOBJN   2,8M
*       50      4M:
                        MOVEM   10,SUM1
*       51              MOVEM   11,SUM2
*       52              MOVEM   12,,R0000
*       53              MOVEM   13,K1
*       54              MOVEM   14,K2
*       55              MOVEM   15,,00001
*       56      2M:
                        MOVEI   2,144
        57              ADDM    2,,R0001
        60              AOS     2,J
        61              AOSGE   0,,S0000
        62              JRST    0,7M
17      63              MOVEI   16,14M
        64              PUSHJ   17,OUI.
        65              MOVEI   16,15M
        66              PUSHJ   17,IOLST.
19      67      10P:
                3M:
                        MOVEI   16,5M
        70              PUSHJ   17,EXIT.
```

```
 ARGUMENT BLOCKS:                                    Argument blocks

        71                      0,,0
        72          5M:         0,,0
        73                      777773,,0
        74          14M:        0,,777775
        75                      0,,0
        76                      0,,0
        77                      340,,116115
        100                     0,,7
        101                     0,,0
        102         15M:        1100,,SUM1
        103                     1100,,SUM2
        104                     4000,,0
MAIN.   0 ERRORS DETECTED                           errors summary
```

B.3 ERROR REPORTING

If an error occurs during the initial pass of the compiler (while the actual source code is being read and processed), an error message referencing the internal sequence number of the incorrect line is printed on the listing following the incorrect line and on the user's terminal. An English text that describes the error follows the message.

Errors that are detected after the initial pass of the compiler appear at the end of the listing (or terminal output). They also reference the internal sequence number of the line in error.

A reference to an internal sequence number of zero (0) is an error detected after the initial pass of the compiler in a line that was inserted by the compiler. The accompanying message, however, accurately describes the error.

There are two levels of error, warning and fatal. The warning message may or may not actually contain the word WARNING. It indicates a minor error or inconsistency. The compilation will continue and the object program may be correct. If a fatal error is encountered in the initial pass of the compiler, the remaining passes will not be called. As the word fatal denotes, it is not possible to generate a correct object program for the source program containing the error.

Most errors (both levels) are detected in the initial pass of the compiler. However, if fatal errors are detected in the initial compiler pass, additional errors that would be detected in later compiler passes may not become apparent until the first errors are corrected.

The printing of error and warning messages on the user's terminal can be suppressed by use of the /NOERRORS or /NOWARNINGS switches, respectively.

At the end of the listing file and on the terminal, an error summary is printed after each subprogram unit is compiled. This message is of the form:

NAME number ERRORS DETECTED

Where name is the subprogram name and number is the cumulative total of all fatal errors and warnings. If this line is followed by a question mark (?) appearing on the far left of the line, fatal errors were encountered.

B.4 WRITING EFFECTIVE FORTRAN-10 PROGRAMS

B.4.1 General Programming Considerations

Programming considerations that should be observed when preparing a FORTRAN program to be compiled by FORTRAN-10 are described in the following paragraphs.

B.4.1.1 Accuracy and Range of Double Precision Numbers – Floating point and real numbers may consist of up to 16 digits in a double precision mode; they must be within the range (decimal) ±.1468E-38 to ±114E+38. Care must be taken when testing the value of a number within the foregoing range since, although numbers up to 10^{38} may be represented, FORTRAN-10 can only test numbers of up to 8 significant digits (REAL precision) and 16 significant digits (DOUBLE precision).

Care must also be taken when testing the floating point computation for a result of 0. In most cases the anticipated result (i.e., 0) will be obtained; however, in some cases the result may be a very small number which approximates 0. Such an approximation of 0 would cause tests within statements (e.g., an arithmetic IF) to fail.

B.4.1.2 Writing FORTRAN-10 Programs for Execution on Non-DEC Machines – If a program is to prepared to run on both a DECsystem-10 computer and a non-DEC machine, the user should:

1. Avoid using the non-standard features of FORTRAN-10, and

2. Consider the accuracy and size of the numbers which the non-DEC machine is capable of handling.

B.4.2 Storage of Arrays

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

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

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

```
   B(1,1,1) → B(2,1,1) → B(3,1,1) --
-----------------------------------
└→ B(1,2,1) → B(2,2,1) → B(3,2,1) --
-----------------------------------
└→ B(1,3,1  → B(2,3,1) → B(3,3,1) --
-----------------------------------
└→ B(1,1,2) → B(2,1,2) → B(3,1,2) --
-----------------------------------
└→ B(1,2,2) → B(2,2,2) → B(3,2,2) --
-----------------------------------
└→ B(1,3,2) → B(2,3,2) → B(3,3,2) --
-----------------------------------
└→ B(1,1,3) → B(2,1,3) → B(3,1,3) --
-----------------------------------
└→ B(1,2,3) → B(2,2,3) → B(3,2,3) --
-----------------------------------
└→ B(1,3,3) → B(2,3,3) → B(3,3,3)
-----------------------------------
```

B.4.3 Use of COMMON

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

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

where: The enclosed letters /A1/, /A2/, and /An/ represent optional name constructs (referred to as COMMON BLOCK NAMES when used).

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

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

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

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

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

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

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

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

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

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

During compilation of a source program, FORTRAN-10 will string together all items listed for each labeled common area and for blank common in the order in which they appear in the source program statements. For example, the series of source program statements:

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

have the same effect as the single statement:

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

All items specified for blank common are placed into one area. Items within blank common are ordered as they are given throughout the source program.

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

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

B.4.4 Use of EQUIVALENCE Statements

Statements of this type are used to:

1. Save storage space. Arrays that are used independently, but not concurrently, may be made equivalent in order to share the same storage locations. For example, the sequence:

   ```
   DIMENSION A (10), B (5,3)
   EQUIVALENCE (A (1), B (1,1))
   ```

   causes arrays A and B to share the same storage locations with B overlapping A.

2. Take advantage of the convenience of arrays in I/O lists and loops, but still retain meaningful names for selected elements. For example, the sequence:

   ```
   DIMENSION RECORD (10), NAME (2)
   EQUIVALENCE (IDENT, RECORD (1)), (NAME (1),RECORD (2))
   ```

   assigns the identification IDENT to the first location of array RECORD. IDENT is maintained as a non-shared location by equivalence arrays NAME and RECORD starting at location 2 of RECORD.

3. To link logically related variables of different types in one array. For example, the program sequences:

```
DIMENSION RECORD (10), NAME (2)
EQUIVALENCE (IDENT, RECORD (1)), (NAME (1),VOLUME)
```

Care must be taken in equivalencing COMPLEX and DOUBLE PRECISION variables which occupy two storage locations each with variables of types which occupy only one storage location.

4. To handle all elements of a multi-subscripted structure simultaneously in a single loop. For example, the program sequence:

```
DIMENSION A(3,4,7), ZERO(84)
EQUIVALENCE (ZERO(1), A(1,1,1))
DO 10 I=1, 84
10 ZERO (1) = 0
```

Zeroes all elements of the array A using the 1-dimensional array ZERO.

B.4.5 Use of ENTRY Statements

FORTRAN-10 provides an ENTRY statement that enables the user to specify additional entry points into an external subprogram. This statement, in conjunction with a RETURN statement, enables the user to employ only one computational routine of a subprogram which contains several relatively independent sections. The form of the ENTRY statement is:

```
ENTRY name(arg1,arg2,...,argn)
```

where:

name is the symbolic name to be assigned the desired entry point

(arg1–argn) is an optional list of dummy arguments. This list may contain:

1. variable names
2. array declarators
3. the name of an external procedure (SUBROUTINE or FUNCTION), or
4. an address constant denoted by either a * or $ symbol

The rules for the use of an ENTRY statement follow:

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

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

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

4. ENTRY statements are non-executable and do not affect the execution flow of a subprogram.

5. An ENTRY statement may not appear in a main program, nor may a subprogram reference itself through its entry points.

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

7. The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in a SUBROUTINE or FUNCTION statement or any other ENTRY statement in the subprogram. However, the arguments for each call or function reference must agree with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that is referenced.

8. Entry into a subprogram initializes the dummy arguments of the referenced ENTRY statement; all appearances of these arguments in the entire subprogram are initialized.

9. A dummy argument must not be used in any executable statement in the subprogram unless it has been previously defined in an executable statement or in the dummy list of an ENTRY, SUBROUTINE, or FUNCTION statement.

10. The value of the function in function subprograms may be returned by using either the function name or an entry name. The type of the function name and the type of the entry name need not be the same.

Example:

The function subprogram

```
        INTEGER FUNCTION IAB(J,K)
101     IAB = J+K
        RETURN
120     ENTRY ISUB(J)
        IF (J>K)101,150
150     IAB = J-K
        RETURN
        END
```

has two entry points, the FUNCTION statement and statement 120. References to these entry points from another program may be

```
      .
      .
      .
TOTAL = IAB(10+P,ITOT)
      .
      .
      .
SUB = ISUB(50)
```

B.4.6 Using Floating Point DO Loops

FORTRAN-10 permits the user to employ non-integer single or double precision numbers as the parameter variables in a DO statement. The primary advantage of the foregoing is to enable the user to generate a wider range of values for the DO loop index variables which may, in turn, be used inside the loop for computations.

B.4.7 Computation of DO Loop Iterations

The number of times through a DO loop is computed outside the loop and remains static for each cycle performed. The formula for the number of times a DO loop is executed is:

DO 10I = m_1 , m_2 , m_3

$$\frac{m_2 - m, + m_3}{m_3} = \text{Number of cycles}$$

The values of the parameters (i.e, m_1 to m_3) may be of any type, however, proper consideration must be given to the foregoing formula, particularly when using logicals.

B.4.8 List-Directed I/O

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

keyword (u,*)l

Example:

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

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

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

1. Each input constant must be of an acceptable type.

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

3. Blanks serve as delimiters and therefore are not significant in any but literal constants.

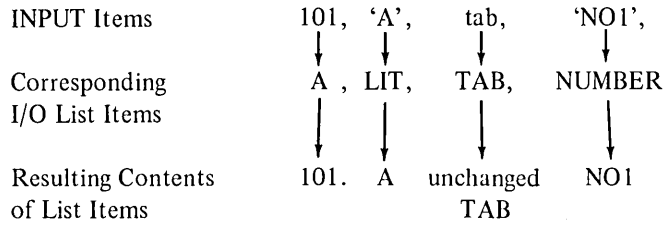
4. Decimal points may be omitted from real constants which do not have a fractional part. FORTRAN-10 assumes that the decimal point follows the rightmost digit of a real constant.

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

1. Delimiters may be either commas or blanks.

2. Delimiters may be either preceded by or followed by any number of blanks, carriage return/line feed characters, tabs, or line terminators; any such combination is considered by FORTRAN-10 as being only a single delimiter.

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



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

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

The repetition of a constant is written as:

r*K

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

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

Examples:

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

B.4.9 Subroutines–Programming Considerations

The following items must be considered when preparing and executing subroutines:

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

2. If the number of actual arguments passed to a subroutine are less than the number of dummy arguments specified, the unused arguments will contain garbage.

3. If the number of actual arguments passed to a subroutine is greater than the number of dummy arguments given, the excess arguments are ignored.

NOTE

No notice is given to the user if either of the situations described in items 1 and 2 occur.

4. If an actual parameter is a constant and its corresponding dummy argument is set to another value, all references made to the constant in the calling program will be changes to the value of the dummy argument.

B.4.10 Reordering of Computations

Computations which are not enclosed within parentheses are reordered by the compiler. Often it is necessary to use parentheses to prevent improper results from being obtained from a specific computation.

For example, assuming that 1) RL_i represents a large number such that RL_i * RL_{i+1} will cause an overflow condition, and that 2) RS_i is a very small number (i.e., less than 1), the program sequence:

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

will not produce an overflow when evaluated in an unoptimized left-to-right manner since the final computation in each expression (i.e., RS1 * RL1 and RS2 * RL2) will produce an interim result which is smaller than either large number (RL1 or RL2).

During optimization of the foregoing sequence, the subexpression RL1 * RL2 is handled as a common subexpression as in the following sequence (optimized):

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

The computation of C will then cause an overflow.

The program sequence should be written in the following manner to ensure that the desired results are obtained after optimization:

```
.
.
A = (RS1 * RL1) * RL2
B = (RS2 * RL2) * RL1
.
.
```

B.5 FORTRAN-10 GLOBAL OPTIMIZER

An optional global[1] object code optimizer may be invoked during compilation. When used, the global optimizer utilizes the output of the lexical and syntax analysis phase of the compiler to develop an optimized source program which is the equivalent of the original program. (The original and optimized programs are considered equivalent if their object programs produce equivalent results and error messages.) The optimized program is then processed through the standard compiler code generation and peephole optimization phase. The function of the global optimizer is to produce an optimized object program which will execute in less time than its equivalent non-optimized object program.

[1] An optimizer which considers groups of statements in the source program as a single entity for optimization purposes is referred to as a global optimizer.

B.5.1 Optimization Techniques

B.5.1.1 Elimination of Common Subexpressions – Often the same subexpression will appear in more than one computation throughout a program. If the values of the operands of such a common expression are not changed between computations, the subexpression may be written as a separate arithmetic expression and the variable representing its resultant may then be substituted where the subexpression appears. For example, the instruction sequence:

```
A = B*C + E*F
      .
      .
      .
H = A + G - B*C
      .
      .
      .
IF ((B*C)-H) 10,20,30
      .
      .
```

contains the common subexpression B*C. Rewriting the foregoing sequence as:

```
T = B*C
A = T + E*F
      .
      .
H = A + G - T
      .
      .
IF ((T)-H) 10,20,30
      .
      .
```

eliminates two computations of the subexpression B*C from the overall sequence.

Decreasing the number of arithmetic operations performed in a source program by the elimination of common subexpressions shortens the execution time of this resulting object program.

The following is a more subtle example of the manner in which the global optimizer applies the foregoing technique. The instructions:

```
A(I,J)=B(I,J)
```

where A and B are dimensioned 25,25 will, during compilation, produce an instruction sequence of the form:

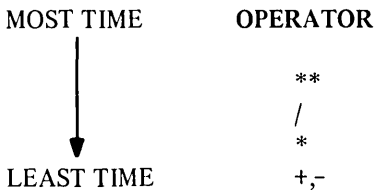
```
K = J*25
K = K + I
Load A(K)
N = J*25
N = N+I
Store B(N)
```

The variables K and N of the foregoing sequence represent equivalent expressions. The global optimizer recognizes this redundancy and will produce the following equivalent sequence:

```
K = J*25
K = K + I
Load A(K)
Store B(K)
```

The optimized sequence will execute faster since it requires the execution of fewer machine instructions, and it requires less internal storage since it is shorter than the original sequence.

B.5.1.2 Reduction of Operator Strength – The time required to execute arithmetic operations will vary according to the operator(s) involved. The hierarchy of arithmetic operators according to the amount of execution time required is:



During the development of an equivalent optimized program, the global optimizer replaces, where possible*, those arithmetic operations which require the most time with operations which require less time. For example, consider the following DO loop which is used to create a table for the conversion of from 1 to 20 miles to their equivalent in feet.

```
      DO 10 MILES = 1,20
10    IFEET(MILES) = 5280*MILES
```

The execution time of the foregoing loop would be shorter if the expensive (in terms of time) multiply operation (i.e., 5280*MILES) could be replaced by a cheaper operation. Since the variable MILES is incremented by 1 on each iteration of the loop, the multiply operation may be replaced by an add and total operation. In its optimized form, the foregoing loop would be replaced by a sequence equivalent to:

```
      K = 5280
      DO 10 MILES = 1,20
      IFEET (MILES) = K
10    K = K + 5280
```

In the optimized form of the loop, the value of K is set to 5280 for the first iteration of the loop and is increased by 5280 for each succeeding iteration of the loop.

The foregoing situation occurs frequently in subscript calculations which implicitly contain multiplications whenever the dimensionality is two or greater.

B.5.1.3 Removal of Constant Computation From Loops – The speed with which a given algorithm may be executed can be increased if instructions and/or computations are moved out of frequently traversed program sequences into less frequently traversed program sequences. Movement of code is possible only if none of the arguments in the items to be moved are redefined within the code sequences from which they are to be taken. Computations within a loop comprised of variables or constants which are not changed in value within the loop may be moved outside the loop. Decreasing the number of computations made within a loop will greatly decrease the execution time required by the loop.

*Numerical analysis considerations severely limit the number of cases where it is possible.

For example, in the sequence:

```
      DO 10 I = 1,100
10    F = 2.0 * Q * A(I) + F
```

The value of the computation 2.0*Q, once calculated on the first iterations, will remain unchanged during the remaining 99 iterations of the loop. Reforming the foregoing sequence to:

```
      QQ = 2.0*Q
      DO 10 I = 1,100
10    F = QQ*A(I) + F
```

moves the calculation 2.0*Q outside of the scope of the loop. This movement of code eliminates 99 multiply operations.

B.5.1.4 Constant Folding and Propagation – In this method of optimization, expressions containing determinate constant values are detected and the constants are replaced, at compile time, by their defined or calculated value. For example, assume that the constant PI is defined and used in the following manner:

```
      .
      .
PI = 3.14159
      .
      .
      .
X = 2*PI * Y
      .
      .
```

At compile time, the optimizer will have used the defined value of PI to calculate the value of the subexpression 2*PI. The optimized sequence would then be:

```
      .
      .
PI = 3.14159
      .
      .
      .
X = 6.28318 * Y
      .
      .
```

thus eliminating a multiply operation from the object code program.

The computation of determinate constant values at compile time is termed "folding;" the use of the defined value of a constant for replacement purposes throughout a program sequence is termed "propagation of the constant." The execution time saved by the foregoing type of compile time optimization is particularly important when the modified instruction occurs in a frequently traversed section of the program.

During the initial compilation procedures, all constants are carried in KI10 processor format to provide maximum accuracy. If a program is being compiled to run on a KA10 processor, the optimizer will cause the double precision constants to be folded and propagated throughout the program at compile time.

B.5.1.5 Removal of Inaccessible Code – The optimizer detects and eliminates any code within the source program which cannot be accessed. In general, the foregoing condition will not exist since programmers will not normally include such code in their programs, however, inaccessible code may appear in a program during the debugging process. The removal of inaccessible code by the optimizer will reduce the size of the optimized object program.

B.5.1.6 Global Register Allocation – During the compilation of a source program the optimizer controls the allocation of registers to minimize computation time in the optimized object program. The intent of the allocation process is to minimize the number of MOVE and MOVEM machine instructions which will appear in the most frequently executed portions of the code.

Allocation is performed on a loop basis working from the inner loop to the outer loop when loops are nested. Each loop is considered only once during the optimization process.

B.5.2 Improper Function References

The ANSI FORTRAN standard prohibits the use of a function's reference that has side effects that will influence the statement in which the function is referenced (such as defining or redefining other elements in the statement). The optimizer depends on strict adherence to the foregoing rule.

If a source program contains a function reference which violates the foregoing rule, it may be compiled without optimization to produce an object program which will give the desired results. The same program when compiled with optimization will produce an object program which may yield results that differ from those produced by the unoptimized object program.

B.5.3 Programming Techniques For Effective Optimization

The following recommendation, when observed during the coding of a FORTRAN source program, will improve the effectiveness of the optimizer.

1. DO loops with an extended range should not be used.
2. When an assigned GO TO statement with an optional label list is used, each label listed should be completely specified.
3. Avoid transferring alphanumeric data directly to or from the fields of a FORMAT statement.
4. Nest loop so that the innermost index varies the most rapidly.
5. In writing nested loops, try to keep the inner loops smaller than 200 lines of code.

B.6 INTERACTING WITH NON–FORTRAN–10 PROGRAMS AND FILES

B.6.1 Calling Sequences

The standard procedures for the writing of DECsystem-10 subroutine calls are described in the following paragraphs.

1. Procedure
    a. The calling program must load the right half of accumulator (AC) 16 with the address of the first argument in the argument list.
    b. The left half of AC 16 must be set to zero.
    c. The subroutine is then called by a PUSHJ instruction to AC 17.
    d. The returns will be made to the instruction immediately after the PUSHJ 17 instruction.

2. Restrictions

   a. Skip returns are not permitted.

   b. The contents of the pushdown stack located before the address specified by AC 17 belongs to the calling program; it cannot be read by the called subprogram.

B.6.2 Accumulator Usage

The specific functions performed by accumulators (AC) 17, 16, 0 and 1 are as follows:

1. Pushdown pointer

   AC 17 is always maintained as a pushdown pointer. Its right half points to the last location in use on the stack and its left half contains the negative of the number of (words–1) allocated to the unused remainder of the stack (a trap occurs when something is pushed into the next to last location. The trap instruction may itself be a PUSHJ on the KI10 processor which uses the last location). A positive left half is not permitted.

2. Argument List pointer

   AC 16 is used as the argument pointer. The called subprogram does not need to preserve its contents. The calling program cannot depend on getting back the address of the argument list he passed to the callee. AC 16 cannot point to the ACs or to the stack.

3. Temporary and value return registers

   AC 0 and 1 are used as temporary registers and for returning values. The called subprogram does not need to preserve the contents of AC 0 or 1 (even if not returning a value). The calling program must never depend on getting back the original contents of the data passed to the called subprogram.

4. Returning values

   At the option of the designer of a called subprogram, a subroutine may pass back results by modifying the arguments, returning a single precision value in AC 0 or a double precision or complex value in AC 0 and 1. A combination of the above may be used. However, two single precision values cannot be returned in AC 0 and 1 since FORTRAN would not be able to handle it.

5. Preserved ACs

   The calling subprogram must preserve ACs 2–15 (octal). Thus, the design of the called subprogram must save and restore any of ACs 2–15 which are changed. This relieves the caller from the burden thereby saving space at the call site. ACs are usually saved on the stack (to save code and data space and allow recursion). However, the allocation of storage for AC saving is up to the called subprogram.

   The design of the called subprogram cannot depend on the contents of any of the ACs being set up by the calling subprogram, except for ACs 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either FORTRAN-10 or COBOL.

B.6.3 Argument Lists

The format of the argument list is as follows:

Arg. list addr.	arg count word
	first arg entry
	second arg entry
	.
	.
	.
	last arg entry

The format of the arg count word is:

bits 0–17 These contain -n, where n is the number of arg entries.

bits 18–35 These are reserved and must be 0.

The format of an arg entry is as follows (each entry is a single word):

bits 0–8 Reserved for future DEC development (set to 0 for now).

bits 9–12 Arg, Type code.

bit 13 Indirect bit if desired.

bits 14–17 Index field, must be 0 for present.

bits 18–35 Address of the argument.

The following restrictions should be observed:

1. Neither the argument lists nor the arguments themselves can be on the stack. This restriction is imposed so that the stack can be moved at any time to its overflow or the overflow of an adjacent region. The same restriction applies to any indirect argument pointers. Furthermore, neither the argument list nor the arguments can be in the ACs.

2. The called program may not modify the argument list itself. The argument list may be in a write-protected segment, but cannot be in the ACs.

   Note that the arg count word is at position −1 with respect to the contents of AC 16. This word is always required even if the subroutine does not handle a variable number of arguments. A subroutine which has no arguments, e.g., RANDOM, must still provide an argument list consisting of one word (i.e., the argument count word with a 0 in it).

Example:

```
        MOVEI   16,1+[EXP -3B17,A,B,C]   ; SETUP ARG LIST
REG
        PUSHJ   17,SUB                   ; CALL SUBROUTINE
        . . .                            ; RETURN HERE
        . . .

; SUBROUTINE TO SET THIRD ARG TO SUM OF FIRST TWO ARGS
SUB;    MOVE    T,@0(16)                 ; GET FIRST ARG
        ADD     T,@1(16)                 ; ADD SECOND ARG
        MOVEM   T,@2(16)                 ; SET THIRD ARG
        POPJ    17,                      ; RETURN TO CALLER
```

B.6.4 Argument Types

Table B-2
Argument Types and Type Codes

Type Code	Description	
	FORTRAN Use	COBOL Use
0	unspecified	unspecified
1	FORTRAN logical	
2	integer	1-word-COMP
3		
4	real	COMP-1
5		
6		
7	label	procedure address
10	double real	
11		2-word COMP
12		
13		
14	complex	
15		byte string descriptor
16		
17	ASCIZ string	

Literal arguments are permitted, but they must reside in a writable segment. This is because FORTRAN must copy all formals back to the caller's arguments in order to conform to the ANSI standard.

All unused type codes are reserved for future DEC development.

B.6.5 Description of Arguments

The types of the arguments which may be passed are:

1. Type 0 Unspecified

   The calling program has not specified the type. The called subprogram should assume that the argument is of the correct type if it is checking types. If several types are possible, the called subprogram should assume a default as part of its specification. If none of the above conditions are true, the called subprogram should handle the argument as an integer (type 2).

2. Type 1 FORTRAN logical

   A 36-bit binary value containing 0 to specify 'FALSE' and non-0 to specify 'TRUE'.

3. Type 2 Integer and 1-word-COMP

   A 36-bit 2's complement signed binary integer.

4. Type 4 Real and COMP-1

   A 36-bit DECsystem-10 format floating point number:

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

5. Type 6 Octal

   A 36-bit unsigned binary value.

6. Type 7 Label and procedure address

   A 23-bit memory address right justified in a 36-bit word:

bits 0–12	always 0
bit 13	indirect flag
bits 14–17	index register
bits 18–35	the basic address

7. Type 10 Double real

   A double precision floating point number for the CPU being executed (i.e., KA format on a KA10 and KI format on a KI10).

8. Type 11 2-word-COMP

   A 2-word (72-bit) 2's complement signed binary integer:

word 1, bit 0	sign
word 1, bits 1–35	high order
word 2, bit 0	same as word 1, bit 0
word 2, bits 1–35	low order

9. Type 12 Double octal

   A 72-bit unsigned binary value.

10. Type 14 Complex

    A complex number represented as an ordered pair of 36-bit floating point numbers. The first of which represents the real part and the second of which represents the imaginary part.

11. Type 15 Byte String descriptor

    The format of the byte string descriptor is:

word 1: ILDB-type byte pointer (i.e., aimed at the byte preceding the first byte of the string)

word 2: EXP byte count

The byte descriptor may not be modified by the called program.

The byte string itself must consist of a string of contiguous bytes of a uniform size. The byte size may be any number of bits from 1 to 36. The byte count must be large enough to encompass 256K words of storage, i.e., 24 bits for 1-bit bytes. The rest of the word must be 0.

12. Type 17 ASCIZ string

A string of contiguous 7-bit ASCII bytes left justified on the word boundary of the first word and terminated by a null byte in the last word. The length of the string may be from 1 to 256K words.

B.6.6 Converting Existing MACRO-10 Libraries for use with FORTRAN-10

The following simple example illustrates the FORTRAN-10 calling sequence.

```
MAIN.   EX1.FOR FORTRAN V.1 /KI 9-NOV-73        12:52   PAGE 1


00001    C       AN EXAMPLE OF A CALL TO A SUBROUTINE WITH A VARIETY OF ARGUMENTS
00002
00003            DOUBLE PRECISION DP
00004            DIMENSION B(10)
00005    C       THE ARGUMENTS ARE:
00006    C       1. A REAL VARIABLE
00007    C       2. AN ARRAY NAME
00008    C       3. AN ARRAY ELEMENT REFERENCE
00009    C       4. AN INTEGER VARIABLE
00010    C       5. A DOUBLE PRECISION VARIABLE
00011    C       6. AN OCTAL CONSTANT
00012    C       7. A LITERAL
00013
00014            CALL SUB1(A,B,B(I),K,DP,"777,'ABC')
00015
00016            END
00017
00018            S

SUBPROGRAMS CALLED

SUB1


SCALARS AND ARRAYS

DP      1       K       3       B       4       A       16      I       17

TEMPORARIES

.Q0000  20
```

```
 LINE     LOC      LABEL    GENERATED CODE

          0                 JFCL     0,0
          1                 JSP      16,RESET.
          2                          0,0
 14       3                 MOVE     15,I          }  Code to set up the array
          4                 MOVEI    15,B-1(15)    }  element reference as an
          5                 MOVEM    15,,Q0000     }  actual argument
          6                 MOVEI    16,2M
          7                 PUSHJ    17,SUB1
 16       10                MOVEI    16,1M
          11                PUSHJ    17,EXIT.

  ARGUMENT BLOCKS:

          12                0,,0      }  Argument block for a call with no arguments
          13       1M:      0,,0      }
          14                777771,,0     word with negative argument count
          15       2M:      200,,A    }  argument type is "real"
          16                200,,B    }
          17                220,,,Q0000   NOTE:  indirect bit set (array element reference)
          20                100,,K        type is integer
          21                400,,DP       type is double precision
          22                0,,[000000000777]   type is unspecified
          23                740,,[406050320100] type is literal

MAIN.    EX1.FOR FORTRAN V,1 /KI 9-NOV-73          12:52    PAGE 1


          SUBROUTINE SUB1(REAL1,ARYNAM,ARYELM,INT1,DBLPRC,OCT,LIT)
00001               DOUBLE PRECISION DBLPRC
00002               DIMENSION ARYNAM(10)
00003
00004     C         THIS SUB-PROGRAM ILLUSTRATES THE USE AND MODIFICATION OF
00005     C         SOME OF THE ARGUMENT TYPES
00006
00007               REAL1=ARYELM
00008               Q=ARYNAM(INT1)
00009               OCT="776
00010               RETURN
00011               END
```

```
SUBPROGRAMS CALLED


SCALARS AND ARRAYS

LIT     1       OCT     2       Q       3       ARYELM  4       DBLPRC  5       REAL1   7
INT1    10      ARYNAM  11

TEMPORARIES

.SUB16  12              0               636542,,210000


LINE    LOC     LABEL   GENERATED CODE
```

A local copy of all parameters is made

```
*
        SUB1:
        0               MOVEM   16,.SUB16
        1               MOVE    0,@0(16)
        2               MOVEM   0,REAL1
        3               MOVEI   0,@1(16)
        4               MOVEM   0,ARYNAM
        5               MOVE    0,@2(16)
        6               MOVEM   0,ARYELM
        7               MOVE    0,@3(16)
        10              MOVEM   0,INT1
        11              DMOVE   0,@4(16)
        12              DMOVEM  0,DBLPRC
        13              MOVE    0,@5(16)
        14              MOVEM   0,OCT
        15              MOVE    0,@6(16)
        16              MOVEM   0,LIT
        17              JRST    0,3M
        20      2M:
                        MOVE    16,.SUB16
        21              MOVE    0,REAL1
        22              MOVEM   0,@0(16)
        23              MOVE    0,OCT
        24              MOVEM   0,@5(16)
```

Note that for the array, it is the address that is copied and stored in ARYNAM

Epilogue:
Only parameters that are changed are copied back.
This includes the constant which is then "clobbered"

```
        25                      POPJ    17,0
7       26          3M:
                                MOVE    2,ARYELM

        27                      MOVEM   2,REAL1
8       30                      MOVE    2,INT1
        31                      ADD     2,ARYNAM
        32                      MOVE    2,777777(2)
        33                      MOVEM   2,Q
9       34                      MOVEI   2,776
        35                      MOVEM   2,OCT
10      36                      JRST    0,2M
11      37                      JRST    0,2M

 ARGUMENT BLOCKS:

        40                      0,,0
        41          1M:         0,,0
SUB1    0 ERRORS DETECTED
```

The formal array (line 8) is referenced by adding the address (passed as an actual argument) to the subscript value and using that as an index.

To conveniently convert existing MACRO-10 programs so that they will still load and execute correctly when called from F40 or FORTRAN-10, the user can:

1. transfer the initial entry sequence for a routine to:

```
entry: CAIA
       PUSHJ 17, entry
```

2. Change all returns to POPJ 17, 0.

These are the functions performed by the HELLO and GOODBY macros. These macros (available in the file FORPRM.MAC, part of the FOROTS release) were successfully used in converting the library routines to run with both F40 and FORTRAN-10.

In addition, since the FORTRAN-10 compiler uses the indirect bits on argument lists (note that this permits shared, pure code argument lists), it is essential that code which accesses parameters takes this into account. Specifically, sequences that obtained the values of parameters through use of operations such as

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

to pick up the second argument may be changed to

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

This latter operation will work when interfacing to either F40 or FORTRAN-10.

Refer to the previous example which illustrates the code generated by the FORTRAN-10 compiler for specific details of how each argument is accessed. Note specifically that in the case of the formal array, it is the address of the array that is accessed through the:

```
MOVEI O, @ X (16)
```

B.6.7 Mixing FORTRAN-10 and F40 Compiled Programs

Starting with Version 1A of LINK-10, use of the switch /MIXFOR will permit loading of FORTRAN-10 and F40 programs. This is achieved by modifying the code while it is loaded.

This introduces extra code that results in a degradation of the execution of programs so loaded. This feature is provided as a convenience for conversion. It is not intended that it be used for other than conversion assistance.

B.6.8 Interaction with COBOL-10

The FORTRAN-10 programmer may call COBOL-10 programs as subprograms and, conversely, the COBOL programmers may call FORTRAN-10 programs as subprograms.

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

B.6.8.1 Calling FORTRAN-10 Subprograms as COBOL-10 Programs – COBOL programmers may write subprograms in FORTRAN-10 to utilize the conveniences and facilities provided by this language. The COBOL verb ENTER is used to call FORTRAN-10 subroutines. The form of ENTER is as follows:

ENTER FORTRAN program name [using { identifier-1 / literal-1 / procedure name-1 } [, { identifier-2 / literal-2 / procedure name-2 }] ...]

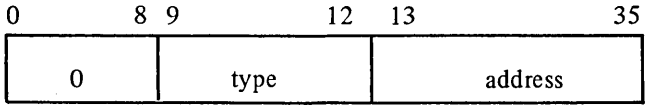
The USING clause of the foregoing forms moves the data within the COBOL program which is to be passed to the called FORTRAN-10 subprogram. The passed data must be in a form acceptable to FORTRAN-10.

The calling sequence used by COBOL in calling a FORTRAN-10 subprogram is:

MOVEI 16, address of first entry in argument list

PUSHJ 17, subprogram address

If the USING clause appears in the ENTER statement, an argument list is created which contains an entry for each identifier or literal in the order of appearance in the USING clause. It is preceded by a word containing, in its left half, the negative number of the number of entries in the list. If no USING clause is present, the argument list contains an empty word and the preceding word is set to 0. Each entry in the list is one 36-bit word at the form:



Bits 0–8 are always 0.

Bits 9–12 contain a type code that indicates the USAGE of the argument.

Bits 13–35 contain the address of the argument or the first word of the argument; the address can be indexed or indirect.

The types, their codes, how the codes appear in the argument list, and the locations specified by the addresses are described below.

a. For 1-word COMPUTATIONAL items

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

b. For 2-word COMPUTATIONAL items

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

c. For COMPUTATIONAL-1 items

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

d. For DISPLAY-6 and DISPLAY-7 items

CODE:	15
IN ARGUMENT LIST:	XWD 640, address
ADDRESS:	that of a 2-word descriptor for the argument
WORD 1:	a byte pointer to the identifier or literal
WORD 2:	bit 0 is 1 if the item is numeric bit 1 is 1 if the item is signed bit 2 is 1 if the item is a figurative constant (including ALL) bit 3 is 1 if the item is a literal bits 4 through 11 are reserved for expansion bit 12 is 1 if the item has a PICTURE with one or more Ps just before the decimal point (e.g., 99PPV) bits 13 through 17 are the number of decimal places. If bit 12 is 1, this is the number of Ps. bits 18 through 35 contain the size of the item in bytes.

e. For procedure names (which cannot be used for calls to COBOL subprograms)

CODE:	7
IN ARGUMENT LIST:	XWD 340, address
ADDRESS:	that of the procedure

The return from a subprogram is POPJ 17 statement after call.

B.6.8.2 Calling COBOL-10 Subroutines From FORTRAN-10 Programs – To call COBOL subroutines use the standard subroutine calling mechanism:

CALL COBOLS (args . . .) subroutine call

or

X = COBOLS (args . . .) function call

The COBOL subroutine must have been compiled using the COBOL compiler described in the *DECsystem-10 COBOL Programmer's Reference Manual.*

B.6.9 FOROTS/FORSE Compatibility

The information presented in Paragraphs B.6.9.1 and B.6.9.2 is intended only for those users who have programs and data files which were developed using the F40 FORTRAN compiler and the FORSE object time system. The manner in which both upward and downward compatibility between the FORTRAN-10, FOROTS and F40, FORSE FORTRAN systems may be achieved is described in the following sections.

B.6.9.1 FORTRAN-10/F40 Data File Compatibility – Upward compatibility of data files (FORSE TO FOROTS) is described in Table B-3. Downward compatibility of data files (FOROTS TO FORSE) is described in Table B-4.

B.6.9.2 Conversion of FOROTS-Developed Data Files Into a Form Acceptable to FORSE – The following paragraphs describe procedures which may be used to convert FOROTS sequential mixed, random access ASCII and random access binary data files into a form which can be read by FORSE.

B.6.9.2.1 Conversion of FOROTS Sequential Mixed Files – The following steps are suggested as a method of converting a FOROTS sequential mixed file into either a sequential ASCII or sequential binary file which is acceptable to FORSE:

1. Prepare and run a FORTRAN-10 I/O program which will produce either a sequential ASCII or a sequential binary output file.

2. If a sequential ASCII file is produced, it must be line-blocked before it can be read by FORSE. Line-blocking is accomplished by copying the file using either the system COPY command (with an A switch) or PIP. The copy will be line-blocked and will be acceptable to FORSE. The following is an example of the command sequence needed to line-block the data file FOROT.DAT:

```
.COPY FOROT.DAT = FOROT.DAT/A
```

3. If a sequential binary file is produced it must be record-blocked before it can be read by FORSE. Record-blocking is accomplished using the /K feature of the program BAKWDS. The following is an example of the command sequence needed to record-block the data file FOROT.DAT:

```
.R BAKWDS
*FOROT.DAT = FOROT.DAT/K
```

Table B-3
Upward Compatibility (FORSE TO FOROTS)

FORSE File Type	May Be Read by FOROTS	In the Following Manner:
1. Sequential ASCII	Yes	May be read directly; record positioning operations (e.g., BACKSPACE, SKIP RECORD) may be used.
2. Sequential Binary	Yes	May be read directly in a forward fashion only, record positioning operations are not permitted.
3. Sequential Mixed Files	Yes	May be read directly in a forward fashion only, record positioning operations not permitted.
4. Random Access ASCII Files	No	NOTE: It is suggested that the random access file be read (using FORSE) and be rewritten as a sequential file which can be accepted by FOROTS.
5. Random Access Binary Files	No	(The conversion suggested in the above note also applies in this case).

Table B-4
Downward Compatibility (FOROTS TO FORSE)

FOROTS File Type	May Be Read by FORSE	In the Following Manner:
1. Sequential ASCII File	Yes	This operation is permitted if the file is line-blocked. This may be accomplished by making a copy of the file using either the system copy command (with an A switch) or the PIP program. The resulting copy will be line-blocked. An example of the command sequence needed to line block a FOROTS file, using PIP, follows: .R PIP *FORSE.DAT=FOROTS.DAT/A
2. Sequential Binary File	Yes	This operation is permitted if the file is record-blocked. This type of blocking is accomplished by using the /K option of the program BAKWDS. The following is an example of a command sequence which record-blocks a file. .R BAKWDS *FORSE.DAT=FOROTS.DAT/K)
3. Sequential Mixed File	No	(See Paragraph B.6.9.2.1 for suggested conversion procedure).
4. Random Access ASCII File	No	(See Paragraph B.6.9.2.1 for suggested conversion procedure).
5. Random Access Binary File	No	(See Paragraph B.6.9.2.1 for suggested conversion procedure).

B.6.9.2.2 Conversion of FOROTS Random Access ASCII Files – The following procedure is suggested as a method of converting a FOROTS random access ASCII file into a form acceptable to FORSE.

1. Prepare and run a FORTRAN-10 I/O program which will create a sequential ASCII file comprised of the records of the random access file.

2. Line-block the sequential ASCII file using either the system COPY command (with an A switch) or the PIP program. The following is an example of the COPY command:

   .COPY LNBLK.DAT = SEQFL.DAT/A

   The foregoing command would produce a line-blocked copy (LNBLK.DAT) of the sequential file SEQFL.DAT.

3. Prepare and run an F40 I/O program which will read the file produced in step 2 and will rewrite the file as a FORSE-generated random access file.

B.6.9.2.3 Conversion of FOROTS Random Access Binary Files – The following procedure is suggested as a method of converting a FOROTS random access binary file into a form acceptable to FORSE.

1. Prepare and run a FORTRAN-10 I/O program which will create a sequential binary file comprised of the records of the random access file.

2. Record-block the sequential file. This is accomplished by using the /K feature of the program BAKWDS. The following example illustrates the command sequence required to convert the file FOROTS.DAT into the record-blocked file FORBLK.DAT.

```
.R BAKWDS
*FORBLK.DAT = FOROTS.DAT/K
```

3. An F40 I/O program may then be written to convert the sequential record-blocked file into a FORSE generated random access file.

B.6.9.3 General Restrictions – The following restrictions must be observed during the preparation of FORTRAN-10 programs and data files.

1. Sets of files comprised of a mixture of files compiled for the KA10 processor and files compiled for the KI10 processor will not be loaded by the LINKING LOADER. If such a set is detected, the LOADER will abort the load operation and will issue the following message at the user's terminal (or log file if in Batch mode):

```
? CANNOT MIX KA10 AND KI10 COMPILED CODE
```

2. CHAIN functions (as implemented for the F40A compiler) are not implemented in FORTRAN-10. An overlay capability which is greatly superior to CHAIN will be implemented in a future version of LINK-10.

APPENDIX C
FOROTS

C.1 INTRODUCTION

The primary function of FOROTS is to act as a direct interface between user object programs and the DECsystem-10 monitor during input and output operations. It implements all program data file functions (e.g., READ, WRITE, REWIND, etc.) and provides the user with an extensive run-time error reporting system. The functions needed to define files, assign devices, allocate core memory for data buffers, and data conversion are all provided by this operating system.

FOROTS implements all standard FORTRAN I/O operations as described in the standard entitled "American National Standard FORTRAN, ANSI X3.9–1966." In addition, FOROTS provides the user with capabilities and programming features beyond those described in the ANSI standard.

C.1.1 Hardware Requirements

FOROTS may be run using either a DECsystem-10 KA10 or KI10 processor. It requires:

a. a minimum of 6K of user core (not including monitor or user program requirements) and

b. re-entrant hardware.

FOROTS interfaces with all DECsystem-10 peripheral devices.

C.1.2 Software Requirements

The software items associated with FOROTS are:

a. a 5.06 or later monitor,

b. the MACRO-10 assembler (Version 47 or later),

c. the LINK-10 loader (Version 1A or later),

d. the system program COMPIL (Version 22 or later), and

e. the FORTRAN-10 Compiler.

C.2 FEATURES OF FOROTS

Many user features of FOROTS are described, briefly, in the following list; more detailed information concerning the implementation of these features is given later in this appendix.

a. A user program may run in either batch or timesharing mode without changing the program. All differences between batch mode and timesharing mode operations are resolved by FOROTS.

b. User programs may access both directory and non-directory devices in the same manner.

c. FOROTS provides complete data file compatibility between all DECsystem-10 devices.

d. Line-blocking, a requirement that each output buffer must contain only an integral number of lines (no lines can be split across output buffers) is not required by FOROTS.

e. Line-numbered files may be read and written.

f. Up to 15 data files may be accessed simultaneously. Any number or all of the open data files may be accessed randomly.

g. FOROTS treats devices located at remote stations similar to local devices.

h. Programs written for magnetic tape operations will run correctly on disk under FOROTS supervision. The commands needed for magnetic tape operations are simulated by FOROTS.

i. Object program device and file specifications may be changed or specified via a FOROTS interactive dialogue mode.

j. Non-FORTRAN binary data files may be read, in image mode, by FOROTS.

k. FOROTS provides interactive program/operating system error processing routines. These routines permit the user to route the execution of the program to specific error processing routines whenever designated types of errors are detected.

l. An error traceback facility for fatal errors provides a history of all subroutine calls made back to the main program, together with the line and statement numbers of the point at which the error occurred.

m. Extensive trap handling for arithmetic functions, including default values and error reports, is provided by FOROTS.

n. ASCII and binary records may be mixed in the same file and both may be accessed in either sequential or random access mode.

o. During random access operations, the user may establish a series of buffers to contain sequential records picked from some selected point within the accessed file. The buffered records may be accessed without requiring physical I/O operations. FOROTS will automatically size the storage buffer to hold one logical record of the accessed file; the user may specify more buffering if desired.

p. FOROTS permits the user program to switch from READ to WRITE on the same I/O device without loss of data or buffering.

q. Data files can be either read or written across more than one unit of a mountable storage medium (i.e., magtape, DECtape). FOROTS will issue the required mount or dismount commands to the system operator whenever a change is required.

r. Although primarily designed for use with a FORTRAN Compiler, FOROTS may also be used as an independent I/O system. FOROTS may be used as an I/O system for MACRO-10 object programs as well as for FORTRAN object programs.

C.3 ERROR PROCESSING

The FOROTS error processing system is given control of program execution whenever a run-time error is detected. This system determines the class of the error and either outputs an appropriate message at the controlling user terminal or branches the program to a predesignated error processing routine.

Error messages may be printed at any of several levels of reporting ranging from a simple octal code to a complete descriptive text. The level of error reporting and/or the classes of error which are to be handled by the program may be selected by the user either in his program or via an interactive dialogue mode of operation.

C.4 INPUT/OUTPUT FACILITIES

FOROTS utilizes monitor-buffered I/O during all modes of access. Display devices are supported in dump mode. Formatted text is handled in ASCII line mode; unformatted files are accessed as FORTRAN binary files.

I/O data channels and access modes are described, individually, in the following paragraphs.

C.4.1 Input/Output Channels

Fifteen software channels (1–15) are available in I/O operations. Software channel 0 is reserved for the following system functions:

1. the printing of error messages, and

2. GETSEG UUO operations (loading and initialization of FOROTS).

Software channels 1 through 15 are available for user program data transfer operations. When a request is made for a data channel, a search table is scanned, starting at channel 1, until a free channel is found. The first free channel is assigned to the requesting program; on completion of the assigned transfer, control of the software channel is returned to FOROTS.

C.4.2 File Access Modes

Data may be transferred between processor storage and peripheral devices in two major modes:

1. Sequential and

2. Random

C.4.2.1 Sequential Transfer Mode – In sequential data transfer operations, the records involved are transferred in the same order as they appear in the source file. Each I/O statement executed in this mode transfers the record immediately following the last record transferred from the accessed source file. A special version of the sequential mode (referred to as Append) is available for output (write) operations. The special Append sequential mode permits the user to write a record immediately after the last logical record of the accessed file. During an Append operation, the records already in the accessed file remain unchanged; the only function performed is the appending of the transferred records to the end of the file.

Transfer modes (other than default mode) must be specified by setting the ACCESS option of an OPEN statement to one of several possible arguments. For the sequential mode, the arguments are:

ACCESS = SEQIN	(sequential read-only mode)
ACCESS = SEQOUT	(sequential write-only mode)
ACCESS = SEQUINOUT	(sequential read followed by a sequential write)
ACCESS = APPEND	(sequential Append mode)

C.4.2.2 Random Access Mode – This transfer mode permits records to be accessed and transferred from a source file in any desired order. Random access transfers, however, may be made between processor core and a device which permits random addressing operations (e.g., disk) and to files which have been set up for random access. Files for random access must contain a specified number of identically-sized records which may be accessed, individually, by a record number.

Random access transfers may be carried out in either a read/write mode or a special read-only mode. The read-only mode is designed to permit a large number of users to read, simultaneously, the same files; this enables a (single) data base to be set up for access by a large group of users. Transfer modes (other than system default mode) must be specified by setting the ACCESS option of an OPEN statement to one of several possible arguments. For the random access mode, the arguments are:

ACCESS = RANDOM (random read/write mode)

ACCESS = RANDIN (random special read-only mode)

C.5 ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS

The types of data files which are acceptable to FOROTS are described, individually, in the following paragraphs.

C.5.1 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters; each set is terminated by a vertical paper-motion character (i.e., Form Feed, Vertical Tab, or Line Feed). All ASCII records start on a word boundary; the last word in a record is padded with nulls, if necessary, to ensure that the record also ends on a word boundary. Logical records may be split across storage blocks. There is no implied maximum length for logical records.

NOTE

On sequential input, FOROTS does not require conformation to word boundaries, it reads what it sees; however, any file that is written by FOROTS will conform to the foregoing format requirements.

C.5.2 ASCII Data Files with Line Sequence Numbers

The addition of line sequence numbers to an ASCII data file is performed according to the following rules:

a. Each line sequence number consists of five ASCII decimal digits which are stored in the first word of the line. Each sequence number is right-justified within the first word location with leading 0s.

b. Bit 35 of the line sequence number word must be set to 1.

c. The first character following a line sequence number is interpreted by FOROTS in the following manner:

   1. On input, if the character is either a tab or a blank, it is skipped and the next character is processed; all other characters are processed.

NOTE

The D delimiter used in BASIC data files is treated as a data character and is passed to the user program.

2. On output, the line sequence number is always followed by a tab. FOROTS always adds line sequence numbers to ASCII output files when in this mode. Sequence numbers start at 1 and are incremented by 1 (default value) unless another increment is specified by the user (via an OPEN statement). The maximum line sequence number is 99999; if this number is reached, the count is restarted at 1.

C.5.3 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data which may be referred to by either a READ or a WRITE statement in the program being executed. A logical record is preceded by a control word and may have one or more control words embedded within it. In FORTRAN binary data files there is no relationship between logical records and physical block sizes. When stored, each logical record follows immediately the previous record on the storage medium. There is no implied maximum length for logical records.

C.5.4 Mixed Mode Data Files

FOROTS permits files containing both ASCII and binary data records to be read. Mixed files may be accessed in either sequential or random access mode. Logical ASCII and binary records have the same format as described in the preceding paragraphs.

C.5.5 Image Binary Files

The image binary data transfer mode is a buffered mode in which data is transferred in a blocked format consisting of a word count located in the right half of the first data word of the buffer followed by n number of 36-bit data words. The devices which permit image binary data transfers and the form in which the data is read or written are:

Device	Data Forms
Card Punch	In image binary mode, each buffer contains three 12-bit bytes. Each byte corresponds to one card column. Since there is room for 81 columns in the buffer (3 X 27) and there are only 80 columns on a card, the last word contains only 2 bytes of data; the third byte is thrown away. Image binary causes exactly one card to be punched for each output. The CLOSE punches the last partial card and then punches an EOF card.
Card Reader	All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte contains column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. Cards are not split between two buffers.
Magnetic Tape	Data appears on magnetic tape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation.
Paper Tape Punch	Binary words taken from the output buffer are split into six 6-bit bytes and punched with the eighth hole punched in each frame. There is no format control or checksumming performed by the I/O routine. Data punched in this mode is read back by the paper tape reader in the same mode.
Paper Tape Reader	Characters not having the eighth hole punched are ignored. Characters are truncated to six bits and packed six to the word without further processing. This mode is useful for reading binary tapes having arbitrary blocking format.
Plotter	Six 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.

C.6 USING FOROTS AS A GENERAL I/O SYSTEM

FOROTS has been designed to lend itself for use as an I/O system for programs written in languages other than FORTRAN. Currently, MACRO programmers may employ FOROTS as a general I/O system by writing simple MACRO calls which simulate the calls made to FOROTS by a FORTRAN compiler. The calls made to FOROTS are to routines which implement FORTRAN I/O statements such as READ, WRITE, OPEN, CLOSE, RELEASE, etc.

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

C.6.1 FOROTS Entry Points

FOROTS provides the following entry points for calls from either a FORTRAN compiler or a non-FORTRAN program:

Entry Point	Function
ALCHN.	Allocate software channels
ALCOR.	Allocate dynamic core blocks
CLOSE.	Close a file
DEC.	DECODE routine
DECHN.	Deallocate software channels
DECOR.	Deallocate dynamic core blocks
ENC.	ENCODE routine
EXIT.	Terminate program execution
FIN.	Input/Output list termination routine
FIND.	Position to the next record (RANDOM ACCESS)
FORER.	Error processor
IN.	Formatted input routine
INIT.	Initialize and assign dynamic core
IOLST.	Input/Output list routine
MTOP.	File utility processing routine
NLI.	NAMELIST input routine
NLO.	NAMELIST output routine
OPEN.	Open a file
OUT.	Formatted output routine
RELEA.	Release a device (CLOSE implied)
RESET.	Job initialization entry
RTB.	Binary input routine
TRACE.	Trace subroutine calls
WTB.	Binary output routine

C.6.2 Calling Sequences

All calls made to FOROTS must be made using the following general form:

```
MOVEI    16,ARGBLK
PUSHJ    17,Entry Point
 return
```

where:

a. ARGBLK is the address of a specifically formatted argument block which contains information needed by FOROTS to accomplish the desired I/O operation.

b. Entry Point is an entry point identifier (see list given in Paragraph C.6.1) which specifies the entry point of the desired FOROTS routine.

With three exceptions, all returns from FOROTS will be made to the program instruction immediately following the call PUSHJ 17, entry point instruction. The exceptions are:

a. an error return to a specified statement number (i.e., READ or WRITE statement ERR= option),

b. an end-of-file return to a statement number (i.e., READ or WRITE statement END= option),

c. a fatal error which returns to the monitor or to a debug package.

Paragraphs C.6.3.1 through C.6.3.12 give the MACRO calls and required argument block formats needed to initialize FOROTS and FOROTS I/O operations.

C.6.3 MACRO Calls for FOROTS Functions

The forms of the MACRO calls to FOROTS which are made by the FORTRAN-10 compiler are described in the following paragraphs. The calls described are identified according to the language statement which they implement. The following terms and abbreviations may be used in the description of the argument block (ARGBLK) of each call.

n = binary count of ASCII characters,

f = FORMAT statement number,

v = the name of an array containing ASCII characters,

list = an Input/Output list,

c = the statement to which control is transferred on an "END OF FILE" condition,

d = the statement to which control is transferred on an "ERROR" condition,

name = a NAMELIST name,

#R = a variable specifying the logical record number where I/O begins for random access mode,

repeat = an integer constant, variable, or expression which specifies the number of times the operation is to be repeated,

* = LIST DIRECTED I/O, the FORMAT statement not used,

type = a variable type specification,

addr = a memory address,

I = bit 13 to specify indirect addressing,

X = bits 14–17 for indexing, MUST BE AC 15 OR LESS AT ALL LEVELS,

arg = the count of the number of arguments in the block,

unit = the FORTRAN logical unit number,

T = type field,

→ = pointer to the second word in the argument block (i.e., the address pointed to by the argument ARGBLK in the calling sequence).

C.6.3.1 Initialization of FOROTS – The RESET. call to FOROTS must precede all other calls to the object time system. The general form of the RESET. call is:

```
JSP     16.RESET.
ARG
 (return)
```

where ARG is in the form:

0------12	13	14------17	18------35
flags	I	X	0

NOTE
The instruction ARG is unused (a zero instruction) at this time; it is provided to implement future features of FOROTS.

C.6.3.2 ENCODE/DECODE Calling Sequences – The ENCODE and DECODE statements and the calling sequence of each are:

```
ENCODE (n,f,v) list
ENCODE (n,f,v, END=c,ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, ENC.
```

and

```
DECODE (n,f,v) list
DECODE (n,f,v, END=c,ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, DEC.
```

Where ARGBLK is:

0------8	9------12	13	14------17	18------35
-6				
Reserved	2	I	X	Character Count (n)
	7	I	X	END = c
	7	I	X	ERR = d
	0	I	X	Format Address (f)
	2	I	X	Format Size (in words)
Reserved	0	I	X	Array Address (v)

C.6.3.3 Formatted/Unformatted Transfer Statements, Sequential Access Calling Sequences – The READ and WRITE statements for formatted sequential data transfer operations and their calling sequences are:

```
READ (u,f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, IN.

and

WRITE (u,f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

Where ARGBLK is:

0——8	9——12	13	14——17	18——35
-5				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
	7	I	X	ERR = d
	0	I	X	Format Address (f)
Reserved	2	I	X	Format Size (in words)

The READ and WRITE statements for unformatted sequential data transfer operations and their calling sequences are:

```
READ (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.

and

WRITE (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

Where ARGBLK is:

0——8	9——12	13	14——17	18——35
-3				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
Reserved	7	I	X	ERR = d

C.6.3.4 NAMELIST Data Transfer Statements, Sequential Access Calling Sequences – The READ and WRITE statements for namelist-directed sequential data transfer operations and their calling sequences are:

```
READ (u, name)
READ (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLI.
```

and

```
WRITE (u, name)
WRITE (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLO.
```

Where ARGBLK is:

0——8	9——12	13	14——17	18——35
-4				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
↓	7	I	X	ERR = d
Reserved	0	I	X	Namelist Address

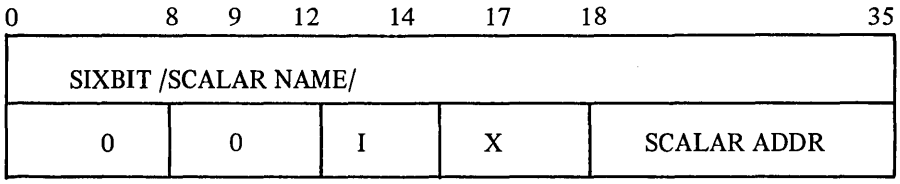
The NAMELIST table illustrated below is generated from the FORTRAN NAMELIST. The first word of the table is the NAMELIST name; following that are a number of 2-word entries for scalar variables, and a number of (N+3)=word entries for array variables, where N is the dimensionality of the array. The NAMELIST argument block has the following formats:

NAMELIST ADDR/

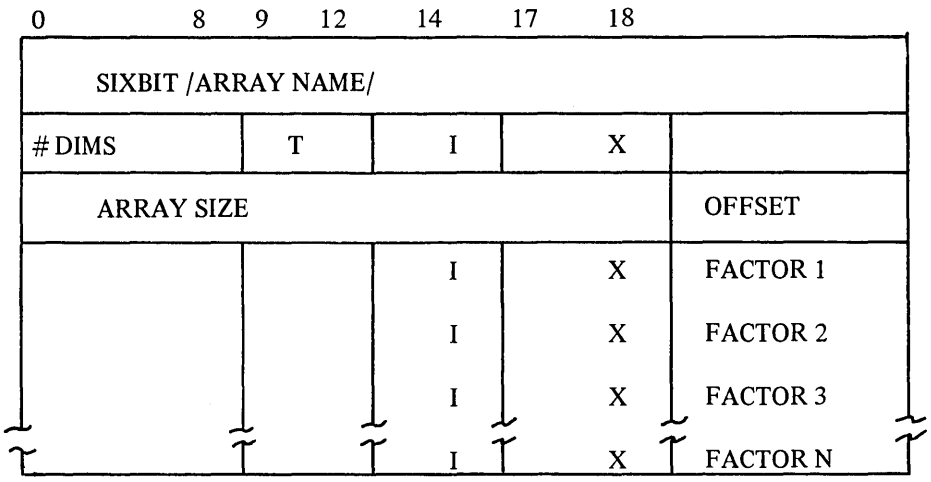
0 8	9 12	14 17	18 35
SIXBIT /NAMELIST NAME/			

The names specified in the NAMELIST statement are stored, in SIXBIT form, first in the table. Each name entry is followed by a list of arguments associated with the name; this argument list may be of any length and is terminated by a 0 entry. The name argument list may be in either a scalar or an array form (refer to the following diagrams).

SCALAR ENTRIES



ARRAY ENTRIES



C.6.3.5 Formatted/Unformatted Data Transfer Statements, Random Access Calling Sequences – The READ and WRITE statements for random access data transfer operations and their calling sequences are:

```
READ (u#R, f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.
```

and

```
WRITE (u#R, f, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

Where ARGBLK is:

0——————8	9——————12	13	14——————17	18——————35
-6				
Reserved	2	I	X	Unit Number (u)
	7	I	X	END = c
	7	I	X	ERR = d
	0			0
	0			0
Reserved	2	I	X	Record Number

C.6.3.6 Calling Sequences for Statements Which Use Default Devices – The FORTRAN-10 statements which require the use of a reserved system default device and their calling sequences are:

	Default Device	
ACCEPT f, list	UNIT=-4	(TTY)
READ f, list	UNIT=-5	(CDR)
REREAD f, list	UNIT=-6	(REREAD)

```
MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

Where ARGBLK is:

0——————8	9——————12	13	14——————17	18——————35
-5				
Reserved	2	I	X	Unit Number (u)
	7	I	X	END = c
	7	I	X	ERR = d
	0	I	X	Format Address (f)
Reserved	2	I	X	Format Size (in words)

	Default Device	
PRINT f, list	UNIT=-3	(LPT)
PUNCH f, list	UNIT=-2	(PTP)
TYPE f, list	UNIT=-1	(TTY)

```
MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

Where ARGBLK is:

0------8	9------12	13	14------17	18------35
-3				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
Reserved	7	I	X	ERR = d

C.6.3.7 Calling Sequences for Statements which Position Magnetic Tape Units – The FORTRAN-10 statements which may be used to control the positioning of a magnetic tape device and their calling sequences are:

COMMANDS:

SKIPFILE u	FUN=7
BACKFILE u	FUN=3
BACKSPACE u	FUN=2
ENDFILE u	FUN=4
REWIND u	FUN=0
SKIPRECORD u	FUN=5
UNLOAD u	FUN=1

CALL:

```
MOVEI 16, ARGBLK
PUSHJ 17, MTOP.
```

Where ARGBLK is:

0------8	9------12	13	14------17	18------35
-3				
Reserved	FUN	I	X	Unit Number (u)
↓	7	I	X	END = c
Reserved	7	I	X	ERR = d

C.6.3.8 List Directed Input/Output Statements – Any form of a formatted input/output statement may be written as a list-directed statement by replacing the referenced FORMAT statement number with an asterisk (*). The list-directed forms of the READ and WRITE statements and their calling sequences are:

```
READ (u, *,END=c, ERR=d) list
READ (u#R, *, END=c ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```
WRITE (u, *, END=c, ERR=d) list
WRITE (u#R, *, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

Where ARGBLK is:

0------8	9------12	13	14------17	18------35
-5				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
	7	I	X	ERR = d
↓				0
Reserved				0

C.6.3.9 Input/Output Data Lists – The compiler generates a calling sequence to the runtime system if an I/O list is defined for the READ or WRITE statement. The argument block associated with the calling sequence contains the addresses of the variables and arrays to be transferred to or from an I/O buffer. The general form of an I/O list calling sequence is:

```
MOVEI 16, ARGBLK
PUSHJ 17, IOLST.
```

Any number of elements may be included in the ARGBLK. The end of the argument block is specified by a zero entry or a call to the FIN. routine.

The elements of an I/O list are:

1. DATA. Value=1

   The DATA. element converts one single, double, or complex precision item from external to internal form for a READ statement and from internal to external form for a WRITE statement. Each DATA. element has the following format.

0——8	9——12	13	14——17	18——35
DATA.	TYPE	I	X	SCALAR ADDR

2. SLIST. Value=2

The SLIST. argument converts an entire array from internal to external form or vice versa depending on the type of statement (i.e., READ or WRITE) involved. An SLIST. table has the following form:

0——8	9——12	13	14——17	18——35
SLIST.		I	X	#ELEMENTS
		I	X	INCREMENT
0	TYPE	I	X	BASE ADDR1.

For example, the sequence:

```
DIMENSION A(100), B(100)
READ (-,-) A
```

develops an SLIST. table of the form:

0——8	9——12	13	14——17	18——35
0				
SLIST.				100
				1
				A

The end of an I/O list is indicated by a call to the FIN. routine in the object time system. This call must be made after each I/O initialization call, including calls with a null I/O list. The FIN. routine may be entered by an explicit call or by an argument in the I/O list argument block. If both calls are used, the explicit call is ignored. The FIN. element has the following formats.

EXPLICIT CALL:

```
PUSHJ 17, FIN.          (FIN.=4)
```

I/O LIST CALL:

0——8	9——12	13	14——17	18——35
4	0	0	0	0

C.6.3.10 OPEN and CLOSE Statements, Calling Sequences – The form and calling sequences for the OPEN and CLOSE FORTRAN-10 statements are:

OPEN STATEMENT CALL

```
MOVEI 16, ARGBLK
PUSHJ 17, OPEN.
```

CLOSE STATEMENT CALL

```
MOVEI 16, ARGBLK
PUSHJ 17, CLOSE.
```

Where ARGBLK is:

0–––––8	9–––––12	13	14–––––17	18–––––35
Negative Count				
Reserved	2	I	X	Unit Number (u)
↓	7	I	X	END = c
↓	7	I	X	ERR = d
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H
G	Type	I	X	H

The G field (bits 0–8) contains a 2-digit numeric which defines the argument name; the H field (bits 18–35) contains an address which points to the value of the argument.

The numeric codes which may appear in the G field and the argument which each identifies are:

G Field	Open Argument	Dialog Argument
01	DIALOG=	/ DIALOG:
02	ACCESS=	/ ACCESS:
03	DEVICE=	/ STRING:
04	BUFFER COUNT=	/ BUFFER COUNT:
05	BLOCK SIZES=	/ BLOCK SIZE:
06	FILE NAME=	/ STRING, STRING:
07	PROTECTION=	/ PROTECTION:
10	DIRECTORY=	[PROJ.,PROG.,SFD, . . .]
11	LIMIT=	/ LIMIT:
12	MODE=	/ MODE:
13	FILE SIZE=	/ FILE SIZE:
14	RECORD SIZE=	/ RECORD SIZE:
15	DISPOSE=	/ DISPOSE:
16	VERSION=	/ VERSION:
17	REELS=	/ REELS:
20	MOUNT=	/ MOUNT:
21	ERROR=	ILLEGAL
22	ASSOCIATE=	ILLEGAL
23	PARITY=ODD, EVEN	/ PARITY:
24	DENSITY=200, 556, 800	/ DENSITY:

C.6.3.11 Memory Allocation Routines – The memory management module is called to allocate or deallocate core blocks. There are two entry points, ALCOR. and DECOR., that control memory allocation and deallocation.

The ALCOR. entry is used to allocate the number of blocks specified by the contents of the argument block variable. Upon return AC 0 will contain either the address of the allocated core block or a -1 value which indicates that core is not available. The calling sequence for ALCOR. call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, ALCOR.
```

Where ARGBLK is:

0–––––8	9–––––12	13	14–––––17	18–––––35
Negative Count				
Reserved	Type	I	X	Number of Words

The DECOR. entry is used to deallocate a previously allocated block of memory; the argument variable must be loaded with the address of the core block to be returned. Upon return, AC 0 is set to 0.

The calling sequence for a DECOR. call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, DECOR.
```

Where ARGBLK is:

0——6	7——12	13	14——17	18——35
Negative Count				
Reserved	Type	I	X	Add of Blocks to be returned

C.6.3.12 Software Channel Allocation and Deallocation Routines – Software channels may be allocated by MACRO programs via calls to the ALCHEN. routine and deallocated by calls to the DECHAN. routine. Values are returned in AC 0.

The ALCHN. entry is used to allocate a particular channel or the next available channel. If the contents of the argument block variable contains a zero the next available channel will be assigned. If the argument block variable is non-zero, it must contain the requested channel number (1–17). If the channel request is not available or all channels are in use, ALCHN. returns with -1 in AC 0. Normal returns contain the assigned channel number in AC 0.

The calling sequence of an ALCHN. routine is:

```
MOVEI 16, ARGBLK
PUSHJ 17, ALCHN.
```

Where ARGBLK is:

0——6	7——12	13	14——17	18——35
-1				
Reserved	Type	I	X	Arg. block variable

The DECHN. entry is used to deallocate a previously assigned channel. The channel to be released is passed to DECHN. in the argument block variable. If the channel to be deallocated was not assigned by ALCHN., AC 0 is set to -1 on return.

The calling sequence for a DECHAN. routine is:

```
MOVEI 16, ARGBLK
PUSHJ 17, DECHN.
```

Where ARGBLK is:

0------8	9------12	13	14------17	18------35
-1				
Reserved	Type	I	X	Channel to be returned

C.7 DETAILED DESCRIPTION

FOROTS is designed to run as either 1) a system comprised of one high segment shared by all users, plus one low segment for each FORTRAN user or 2) a non-sharable system which can be loaded into a user's low segment as part of the user object program. Different users may employ different FORTRAN Object Time Systems within the DECsystem-10 timesharing systems at the same time.

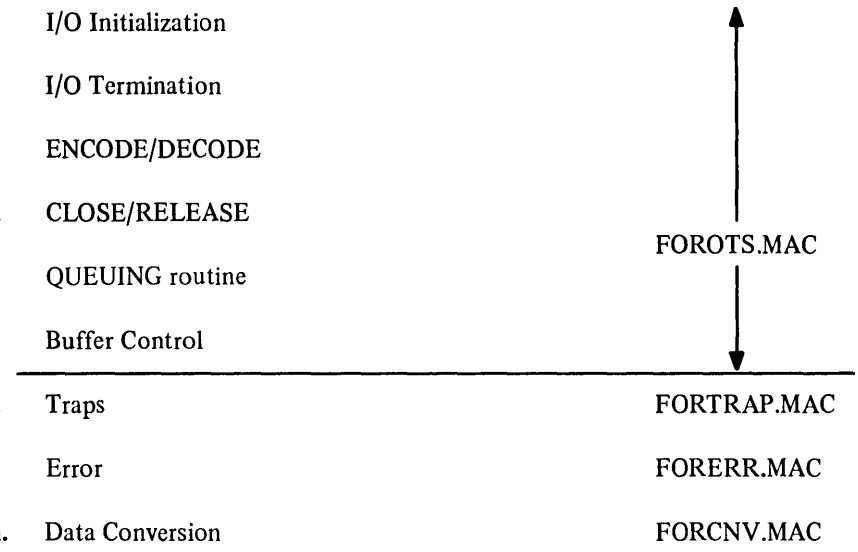
The FOROTS source files are written in MACRO-10 and are part of the FORTRAN library (FORLIB.REL). The source files which compose the FOROTS system are:

Source File	Function
FORPRM.MAC	Parameter File.
FORINI.MAC	Initialization File.
FORCNV.MAC	Format Conversion File.
FORTRP.MAC	Trap Handler
FORJAK.MAC	Translator For Interfacing With the FORTRAN F40 Compiler.
FOROTS.MAC	Main I/O Processing and Control File.
FORERR.MAC	Error Processing File.

More detailed descriptions of the foregoing source files are given in Paragraphs C.7.1.1 through C.7.1.7.

A modular structure was used in designing FOROTS to provide flexibility and ease of maintenance. The correspondence between the FOROTS high-segment modules and its source files is shown in the following table:

MAJOR MODULES	SOURCE FILE
a. Memory Management	
b. OPEN Statement	
c. FORMAT Statement Encoding Routine Dispatch Routine	
d. I/O LIST Routine DATA SLIST ELIST XLIST	FOROTS.MAC



The FOROTS source files not shown in the foregoing table (i.e., FORJAK and FORINI) operate in the user's low segment.

C.7.1 FOROTS Source Files

The source files which contain the modules of the FOROTS systems are described, individually, in the following paragraphs.

C.7.1.1 FORPRM Parameter File – The FORPRM file contains the definitions of, a) accumulator usage, b) pointers to dynamic core blocks, and c) the entry points for the FOROTS high segment. This file is a universal file which must be assembled with all modules of the FOROTS system.

C.7.1.2 FORINI Initialization File – This routine operates in the user's low segment. It checks for the presence of a current FOROTS system in the high segment of core. If a FOROTS system is present, it initializes FOROTS for the segment it represents; if a FOROTS system is not present, it loads the desired sharable version of FOROTS. FORINI is also used during library search operations to resolve any undefined external symbols since it contains the entry point (RESET) which is declared as an EXTERN by all FORTRAN main programs. The file FORINI is called by each RUN, START, or EXECUTE command.

C.7.1.3 FORCNV Data Conversion File – The FORCNV high segment module contains all of the routines required to convert I/O list items from an internal-to-external form and, conversely from an external-to-internal form. The conversion performed is determined by either a FORMAT statement or a variable type code. The FORMAT statement directives always take precedence over a variable type directive when both occur at the same time.

Whenever NAMELIST, LIST DIRECTED, or G descriptors are used to indicate an I/O conversion, the variable type is used to select the conversion routine. Whenever a READ or WRITE statement is used to initiate an I/O operation, the FORCNV conversion type (i.e., routine) is selected from the referenced FORMAT statement.

C.7.1.4 FORTRP Trap Handler – FORTRP is an arithmetic-interrupt routine which, initially, receives control from the FOROTS RESET module. All traps which occur when a job is running are enabled; this includes traps for 1) illegal memory references, 2) nonexistent memory references, 3) pushdown list overflows, 4) integer arithmetic overflow faults, and 5) floating point overflow and underflow faults. Whenever an instruction is detected which causes a trap, the handler interrupts the execution of the program and:

a. for arithmetic faults, it attempts to correct the faulty item by either patching or denormalizing it,

b. outputs an error message if it is required,

c. restarts the program at the instruction immediately following the one that causes the trap if the error condition permits.

C.7.1.5 FORERR Error Routine – This routine, on the detection of a run-time error, determines the class of each error and outputs, if directed, an error message to the controlling user terminal. FORERR is given program control from the user program, FORTRAN library, or the Object Time System whenever a run-time error is detected.

On completion of the error processing procedures, program control is returned to either the user or the monitor depending on the class of the detected error and parameters established for error processing by the user. The user can determine the operations to be performed for specific classes of errors by using error processing routines, the READ or WRITE statement ERR=n argument, and the OPEN statement ERROR argument. The ERR=n argument specifies the address of the first statement of a user routine to which the program is branched whenever an error is detected by FOROTS; the ERROR argument identifies the type and class of the error detected to permit the user routine to determine what action to take.

Text for short error messages to be output to the controlling user's terminal is stored in FORERR; text for long error messages is stored on the systems device.

C.7.1.6 FOROTS Main I/O Processing and Control File – This file contains, in individual modules, the routines needed for:

a. the management of memory during I/O operations,

b. the implementation of the OPEN and FORMAT statement operations,

c. the implementation of the READ, WRITE, and NAME LIST data list processing,

d. the initialization and termination of I/O operations,

e. ENCODE/DECODE functions,

f. queue control,

g. I/O buffer control, and

h. the implementation of both sequential and random access READ and WRITE statements.

The input and output operations performed by FOROTS involve the following five sections of the FOROTS file code:

a. Open Section – This section defines the file and the FORTRAN logical device unit number which are to be read or written.

b. Positioning Section – This section is used during random access to compute a block number from a given logical record number.

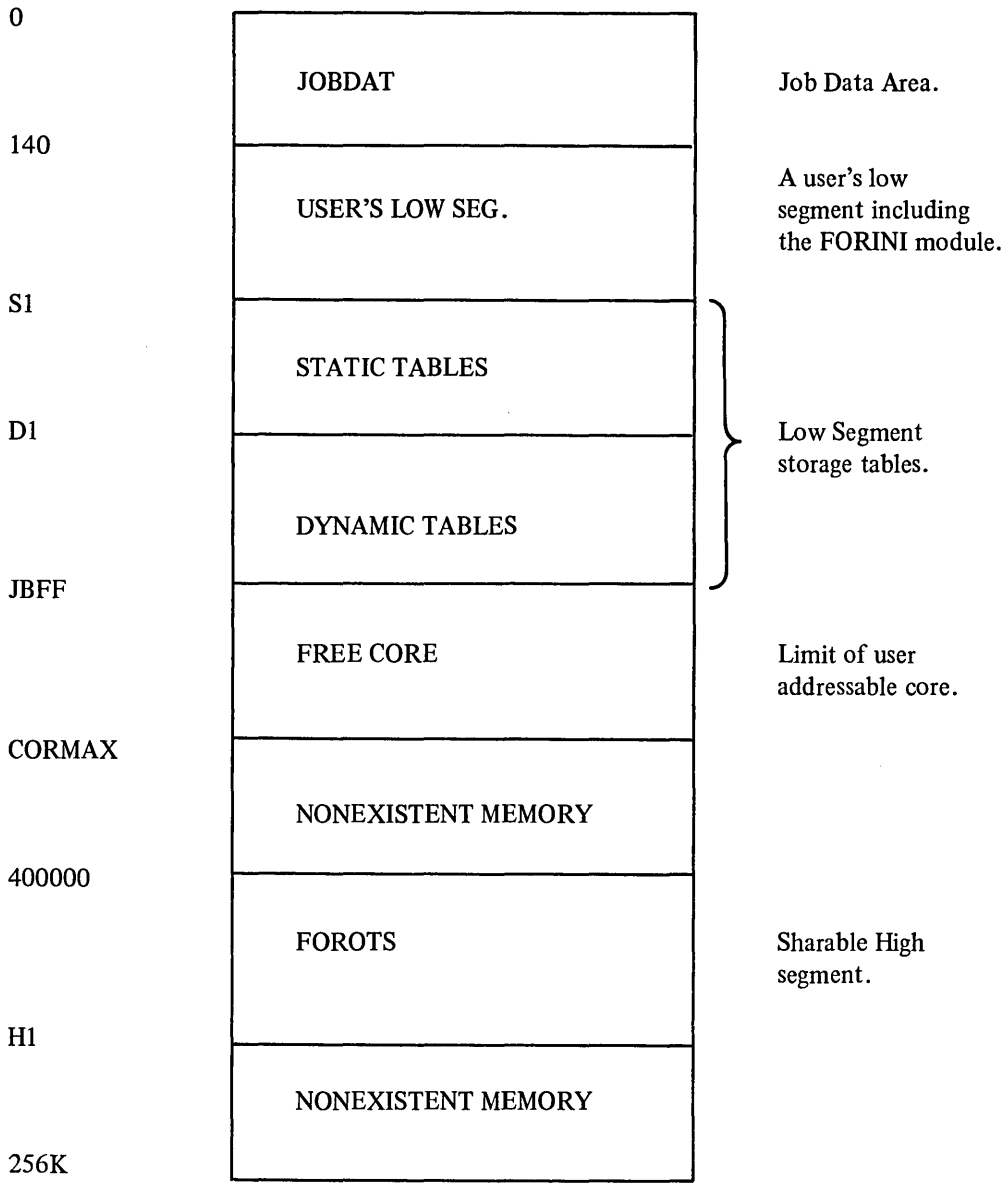
c. I/O List Section – This section controls the transfer of data from an input buffer to the storage locations specified by the variables of an input list or from storage locations specified by an output list to output buffer.

d. FORMAT Statement Section – This section performs the operations needed to convert I/O list variables according to the basic field descriptors given in the FORMAT statement.

e. Closing Section – This section terminates the current I/O operation and disassociates the file involved from the specified FORTRAN logical unit.

C.8 FOROTS CORE REQUIREMENTS

The following core map illustrates a typical core layout during FOROTS operations.



The management of each user's low segment is under the control of the high segment Memory Management Module. The size of a user low segment depends on the I/O activity of the user's object program. The low segment consists of a basic data base, a static table storage area and a dynamic storage area (to location .JBFF).

The static table storage area contains data save areas; tables for ENCODE and DECODE array pointers, logical unit number, software channel number, and an error table; a pointer to FORMAT statement information located in the dynamic storage area and byte pointer. The dynamic core area starts immediately after the static storage area and increases upwards until either the dynamic core requirements are satisfied or the user-addressable space is exhausted (CORMAX is reached).

The sharable high segment of FOROTS occupies less than 6K of core. The modules which comprise the high segment were introduced and described in Paragraphs C.7 through C.7.1.1.

C.8.1 Core and Data File Protection

FOROTS conforms to the standard relocation and protection scheme used by the DECsystem-10 monitors. The sharable high segment of FOROTS is write-protected to prevent terminal user modifications being made to the system. Transferred data files are given the protection code established as the installation standard (default) code unless a code is specified by the PROTECT option of the OPEN statement.

C.9 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

FORTRAN logical and physical device assignments are made by the user at run time or standard system assignments are made according to a FOROTS Device Table (i.e., DEVTB.). The standard assignments contained by the Device Table are shown in Table C-1.

Table C-1
FORTRAN Device Table

Device/Function	FORTRAN Logical Unit Number	Use
REREAD	-6	REREAD statement
CDR	-5	READ statement
TTY	-4	ACCEPT statement
LPT	-3	PRINT statement
PTP	-2	PUNCH statement
TTY	-1	TYPE statement
0	00	ILLEGAL
DSK	01	DISK
CDR	02	Card Reader
LPT	03	Line Printer
CTY	04	Console Teletype
TTY	05	User's Teletype
PTR	06	Paper Tape Reader
PTP	07	Paper Tape Punch
DIS	08	Display
DTA1	09	DECtape
DTA2	10	↓
DTA3	11	
DTA4	12	
DTA5	13	
DTA6	14	
DTA7	15	DECtape
MTA0	16	Magnetic Tape
MTA1	17	↓
MTA2	18	
FORTR	19	Assignable Device
DSK	20	DISK
DSK	21	↓
DSK	22	
DSK	23	
DSK	24	
DEV1	25	Assignable Devices
DEV2	26	↓
DEV3	27	
DEV4	28	
DEV5	29	

APPENDIX D
DEBUGGING FORTRAN PROGRAMS

FORDDT is an interactive program that is used to debug FORTRAN programs. With FORDDT, the user can:

a. change the contents of a variable,

b. set up to 10 pauses in a program,

c. continue from a pause or any other point,

d. type the contents of a variable,

e. display the current pause settings,

f. trace subroutine calls, and

g. display all variables (symbols) in the program.

D.1 LOADING AND STARTING FORDDT

FORDDT is loaded and started with a compiled program by means of the DEBUG command. The user should note that DDT (the standard system debugging program) will also be loaded so that he can also use it for debugging. The command to load and start FORDDT is shown below.

```
.DEBUG file.ext , FORDDT
```

FORDDT responds with the message:

```
>>ENTERING FORDDT
```

Angle brackets (») are used by FORDDT to indicate that it is ready to receive a command. The user can issue any FORDDT command described in the following section.

D.2 FORDDT COMMANDS

Described below are the commands to FORDDT. Only enough letters of the command to make it unique need be typed for FORDDT to recognize the command. Parameters to some commands can be gathered into groups so that the user need not type these parameters each time he wishes to use them. This is accomplished by means of the GROUP command, described below. Unless variables are globally defined within the program, the user can only refer to local variables within the main program or subprogram in which he is currently working. To open a section (main program or subprogram) that is not currently open, the user issues the OPEN command described below. There is no capability in FORDDT to refer to elements in multi-dimensional arrays because the dimension bounds are not

available at run-time. The user must request the appropriate array element. For example, in the multi-dimensional array A (1/5, 1/6), the user must refer to A (2,4) as A (17). He can derive the correct element by means of the formula n+5*(m-1) where n and m are the subscripts. For example, A (2,4)=2+5*(4-1)=17. Also, DO loop parameters are not accessible within the range of the loop because at run-time these parameters are maintained in registers and not in the variables in the DO statement. The user can allow for this by including an assignment in his DO loop which stores the value of the parameter in a variable.

D.2.1 Starting the Program

The START command is used to start or restart execution of the program. The START command has the forms:

```
START
START n
START program name
```

The START command without an argument causes the program to be started or restarted at the beginning. If the START command has a numeric argument, the program will be started at the specified statement label. If the command has a program name as its argument, FORDDT will start the named program at its beginning.

Examples:

```
>>START

>>START 100

>>START MYPROG
```

D.2.2 Stopping the Program

The STOP command terminates program execution, causes all files that are open to be closed, and exits to the monitor. The form of the STOP command is:

```
STOP
```

Example:

```
>>STOP
.EXIT
```

D.2.3 Opening Subprograms

The OPEN command allows the user to open a particular subprogram of the loaded program so that the variables local to the subprogram will be available. When a subprogram is opened, the previously open subprogram or the main program is automatically closed. Only global variables and those variables defined within the currently open subprogram are accessible at any given time. When FORDDT is entered, the main program is automatically opened. The forms of the OPEN command are:

```
OPEN name
OPEN
```

The first form causes the named subprogram to be opened. The second causes the main program to be reopened.

Examples:

```
>>OPEN SUB1
>>OPEN
```

D.2.4 Changing the Values of Variables

The ACCEPT command allows the user to change the contents of a variable in the currently open section of the program. The form of the ACCEPT command is:

ACCEPT variable-name/x value

where:

variable-name is the variable to be changed and

x is the mode of the value. It can have one of the forms shown in the table below.

value is the new value of the variable

Table D-1
Modes for ACCEPT Values

Mode	Meaning	Example
A	ASCII	/FOO/
C	Complex	1.25, 79.01E−5
D	Double Precision	123.4567899
E	Floating Point	123.45678
I	Integer	123456789
O	Octal	7654321
S	Symbolic	PSI (M)

Notes:

1. Only the first five characters between the slashes are used in an ASCII value.
2. When a symbolic value is entered, the current values are used, e.g., in the above example if M = 3, the variable is set to the value in PSI (3).

Examples:

```
>>ACCEPT IPROJ/A  /ELEC./
IPROJ    = -0.2466446E+32          -31354338468.    < E >< L >< E >< C >< . >

>> ACCEPT S/F 3.5
S        =  3.500000               17565745152.
```

D.2.5 Grouping Parameters for Commands

The GROUP command allows the user to store a group of parameters for use in the TYPE or PAUSE commands. This facility eliminates the necessity for the user to type the same parameters in successive TYPE or PAUSE commands. The GROUP command has the forms:

GROUP n list
GROUP n

where:

n is an integer from 1 to 8 that names the group, and
list is any combination of variables, ranges of variables, and other groups.

The form of the GROUP command that contains a list causes the components of the list to be included in the named group. The other form causes the contents of the group to be displayed. If a group is to be included in another group, it must be preceded by a slash (/).

Examples:

```
>>GROUP  6  A, ALPHA(7)-B(7), PI, /5
```

This command causes the variable A, the range of values ALPHA (7) through B (7), the variable PI, and group 5 to be stored as group 6.

```
>>GROUP 6
A, ALPHA(7)-B(7), PI, /5
```

This example causes the contents of group 6 to be displayed.

D.2.6 Specifying Typeout Modes

The MODE command controls the modes in which variables will be typed. Normally each variable will be typed in floating point and integer form. The user can change the modes by means of the MODE command. The forms of the MODE command are:

MODE list
MODE

where list contains one or more modes taken from the following table.

Table D-2
Typeout Modes

Mode	Meaning
F	Floating Point
D	Double Precision
I	Integer
O	Octal
A	ASCII

The second form of the MODE command causes the default mode setting (floating point and integer) to take effect after the mode had previously been changed. The first form sets the mode to those indicated. If all five modes are specified, the modes of the values of the variable are typed in the following order:

Floating point, double precision, integer, octal, ASCII.

Example:

```
>>MODE F, I
>>TYPE I
I         = 0.0000000E-38   1.       <NUL><NUL><NUL><NUL><NUL>
```

D.2.7 Displaying Values

The TYPE command causes the contents of one or more variables in the currently opened section to be typed on the user's terminal. The forms of the TYPE command are:

TYPE list
TYPE

where list contains one or more variable or array names and/or group numbers separated by commas. Group numbers must be preceded by a slash (/).

If an array is specified, the user must specify the exact location of the elements, e.g., B(17) – B(25). This will cause the range of values from B(17) through B(25) to be typed. The variables will be typed in the order specified in the TYPE command. If no variable-names or group-names are specified, FORDDT uses the names specified in the last TYPE command. To request typeout of the value of a complex variable, the variable-name must be specified as an array, e.g., C(1) – C(2).

Examples:

```
>>TYPE L,PRL

L         =  0.0000000E-38   10.

PRL       =  0.1500000      16991964365.
```

D.2.8 Setting Pauses (Breakpoints)

The PAUSE command sets a pause (or breakpoint) at a statement number or subprogram entry point in the program. Up to 8 pauses can be set. The forms of the PAUSE command are:

PAUSE n
PAUSE n AFTER integer
PAUSE n IF condition
PAUSE n TYPING group-number
PAUSE n AFTER integer TYPING group-number
PAUSE n IF condition TYPING group-number

The PAUSE n command causes the program to stop and return to FORDDT each time the numbered statement or named subprogram is encountered. When this occurs, FORDDT types the message:

PAUSE AT n
»

The PAUSE n AFTER integer command causes the program to stop and return control to FORDDT when the statement or subprogram entry point has been passed the specified number of times. The parameter AFTER cannot be abbreviated.

The PAUSE n IF condition command causes the program to stop and return control to FORDDT at the specified statement or subprogram if the specified condition is true. The allowable conditions are .LT., .LE., .GE., .GT., .EQ., and .NE.. The variables and/or numeric values in an IF condition must be of the same type; except that an integer variable can be compared to an ASCII or octal value.

The PAUSE n TYPING group-number command causes FORDDT to suspend program execution temporarily at the specified statement or subprogram entry point and to type the information requested in the specified group. The TYPING parameter cannot be abbreviated.

The PAUSE n AFTER interger TYPING group-number command causes the information requested in the specified group to be typed every time the specified statement or subprogram is encountered and causes the program to stop and return control to FORDDT after the statement or subprogram has been passed the specified number of times.

The PAUSE n IF condition TYPING group-number command causes the information requested by the specified group to be typed every time that the specified statement or subprogram is encountered and causes the program to stop and return control to FORDDT at the specified statement or subprogram if the specified condition is true.

When the PAUSE command is used with the TYPING argument, control will be transferred to FORDDT on the next occurrence of the specified statement if a character is typed on the user's terminal. Thus, typing ahead is not permissible.

Examples:

```
>>PAUSE 30
>>PAUSE 100      AFTER 39
>>PAUSE 55       IF X.LE.Y
>>PAUSE BIV      TYPING 4
>>PAUSE 77       AFTER 14 TYPING 6
>>PAUSE 28       IF A (7) .GE. 3.142E-5 TYPING 3
```

D.2.9 Removing Pauses (Breakpoints)

The RESET command removes the pause (or breakpoint) at a specified line number in the currently open section. The RESET command has the forms:

```
RESET n
RESET
```

If the line number is not specified, all pauses that have been set in the program are removed.

Examples:

```
>>RESET 100
>>RESET
```

D.2.10 Continuing After a Pause (Breakpoint)

The CONTINUE command causes the program to continue execution after a pause (or breakpoint) occurred. The CONTINUE command has the forms:

```
CONTINUE
CONTINUE n
```

where n is a number or variable that will be treated as an integer value.

After a CONTINUE command is executed, the program runs either to completion or until another pause is reached. If a value is included with the command, the program runs until the n^{th} occurrence (proceed count) of the preceding pause has been reached. Note, however, that the value included with the CONTINUE command will be ignored if the pause is conditional (i.e., the PAUSE command contains an IF condition) because conditional pauses are always in effect until reset or redifined.

Example:

```
>>CONTINUE 20
```

D.2.11 Obtaining Information

The HELP command causes typeout of a list of FORDDT commands with examples. No description of command usage is given. The HELP command has the form:

```
HELP
```

Example:

```
>>HELP

COMMANDS:-
PAUSE   RESET   OPEN    START   STOP    CONTINUE        TYPE    ACCEPT
GROUP   WHAT    DDT     LOCATE  STRACE

EXAMPLES
PAUSE   10 AFTER 20 TYPING 2
PAUSE   SUB IF I .GT. A(19) TYPING 3
TYPE    A(B(M)),B(9)-B(5),/4,100,I
ACCEPT  100 ('FORMAT')
ACCEPT  X/COMPLEX 1.2,3.4
GROUP   5 A,I,/3,TAB(24)
MODE    FLOATING,DOUBLE,INTERGER,OCTAL,ASCII
CONTINUE        A(8)
```

The WHAT command causes FORDDT to list the current status of all the user's pause requests and group settings. The form of the WHAT command is:

```
WHAT
```

The output from the WHAT command has the following form:

```
OPEN SECTION: – name
GROUP #PROCEED PAUSE
number { number } { statement number }   IF condition
       {   if   } { subroutine name  }
GROUP #1. variables
          .    .
          .    .
          .    .
GROUP #n
```

If a PAUSE command includes the conditional option, IF is typed under PROCEED and the IF condition is added to the end of the line.

Example:

```
>>WHAT
OPEN SECTION:-FATAN

GROUP#   PROCEED PAUSE
8.       IF      1        IF XOX IN BIV .NE. ALPHAR IN MAIN PROGRAM
         IF      BIV      IF PSI(5.) .GE. -.10345669E-3
4.       100.    802 IN MAIN PROGRAM
6.       0.      804 IN MAIN PROGRAM
         1.      20 IN MAIN PROGRAM


GROUP# 1.
GROUP# 2.
GROUP# 3.
GROUP# 4.
GROUP# 5.
GROUP# 6.
GROUP# 7.
GROUP# 8.
>>
```

The LOCATE command causes typeout of either all variables or the subprogram in which a particular variable is defined. The forms of the LOCATE command are:

```
LOCATE variable
LOCATE
```

The first form of the command is used to obtain the name of the subprogram in which the variable is defined. The second form is used to obtain a list of all global variables available to the program including those from JOBDAT, FOROTS, and the program itself.

Examples:

```
>>LOCATE RJ
MAIN PROGRAM    RIRR
>>LOCATE

PAT..
PAT..

EXIT    CEXIT.
FORXIT

TRACE
TRACE

ALOG10  ALOG.   ALOG    ALG10.
ALOG

EXP.   EXP
EXP

EXP2.6  EXP2.4  EXP2.2  EXP2.0  EXP2.
EXP2

STOP.   PAUSE.  PAUS.
FORPSE

TY%DDB  TRACE%  FORER%
FORERR

NMLTBL  NMLST%  NLOEN.  NLIS0
NMLST%

LSTDR%
LSTDR%

PTLEN.  LOTEN.  HITEN.  EXP10.
POWTB%

OCTAL%
OCTAL%

LOGIC%
LOGIC%

INTEG%
INTEG%

FLOUT%
FLOUT%
```

```
REAL%   FLIRT%
FLIRT%

ALPHA%
ALPHA%

FORQU%
FORQUE

WPOINT  SMEM%%  SAVE.   RELE%%  PMEM%%  OPOINT  OP.SWT  OP.DSP  OBYTE
OBLOK.  NXTLN.  MOD.TB  IPEEK.  IOLS%%  INIT%   IBYTE.  IBLOK.  GMEM%
FOROT%  EXIT%   DPOINT  DIS.TB  CLOSO.  CLOSI.  CLOSB.  ACC.TB
FORCTS

WTB.    VERINI  TRACE.  RTB.    RESET.  RELEA.  OUT.    OPEN.   NLO.
NLI.    MTOP.   IOLST.  IN.     FORER.  FIND.   FIN.    EXIT.   ENC.
DECOR.  DECHN.  DEC.    CLOSE.  ALCOR.  ALCHN.
FORINI

..PETH  RETURN  FORDDT  DDTEND
FORDDT

.Q0000  12      13      14      15      16      17      117     18
19      20      21      22      210     100     211     1       101
2       102     3       103     4       104     400     5       105
6       7       8       108     9       220     109     10      221
11      C       I       L       RNPV    R       R1      IT      .S006
.S0001  J       CONT    SNPV    S1      COC     S2      IPROJ
MAIN PROGRAM

.JBVER  .JBJJO  .JBUSY  .JBTPC  .JBSYM  .JBSA   .JBREN  .JBREL  .JBPF
.JBOPS  .JBOPC  .JBINT  .JBHVR  .JBHSM  .JBHRL  .JBHNM  .JBHGH  .JBHD
.JBHCR  .JBFF   .JBERR  .JBDDT  .JBDA   .JBCST  .JBCOR  .JBCNI  .JBCN
.JBCHN  .JBBLT  .JBAPR  .JB41   %JOBDT  JOBVER  JOBUUO  JOBUSY  JOBTP
JOBSYM  JOBSA   JOBREN  JOBREL  JOBPFI  JOBOPC  JOBINT  JOBHVR  JOBHS
JOBHRL  JOBHNM  JOBHGH  JOBHDA  JOBFF   JOBERR  JOBDDT  JOBDA   JOBCO
JOBCNI  JOBCN6  JOBCHN  JOBBLT  JOBAPR  JOB41
JOBDAT
```

D.2.12 Tracing Subroutine Calls

The STRACE command allows the user to obtain the present nested level of subprogram calls when a pause occurs. The form of the STRACE command is:

STRACE

The output printed is the same as that given by the TRACE option of the PAUSE statement in the FORTRAN-10 compiler. Refer to Paragraph 9.7.1 for a description and example of the printout.

D.2.13 Entering and Leaving DDT

The DDT command allows the user to transfer control from FORDDT to DDT (the standard system debugging program). The form of the DDT command is:

DDT

Example:

```
>>DDT
```

The RETURN command allows the user to return from DDT. The RETURN command has the form:

RETURN (ALTMODE) G

where: (ALTMODE) means ALTMODE or ESCAPE

The return is made to the point in the program where processing had left off due to the transfer to DDT.

Examples:

```
RETURN(ALTMODE) G
```

INDEX

A

A (alphanumeric) field descriptor, 13-9

Access,

- OPEN/CLOSE statement
- option, 12-2

Accuracy and range of double precision numbers, B-8

Action of field descriptors, 13-5

Actual and dummy arguments, agreement between, 15-1

Actual arguments

- CALL statement, 15-10
- external function reference, 15-12
- generic function names, 15-6
- use of, 15-1

Acute accent, 2-2

Adjustable dimensions, 6-2

- type statement, 6-4

Alphanumeric character transfer, 13-10

Alphanumeric field descriptors, 13-9, 13-11

Apostrophe representation, 13-11

Argument lists, B-21

Argument types, B-22

Arguments

- actual, 15-1
- actual function reference, 15-12
- agreement between actual and dummy, 15-1
- description of, B-22
- ENTRY statement, 15-13

Arithmetic assignment statement, 8-1

Arithmetic expression,

- compound, 4-1
- rules for, 4-2
- simple, 4-1

Arithmetic IF statement, 9-3

Arithmetic operations and operators, 4-1

Arrays

- adjustable dimensions, 6-2
- description, 3-7
- dimensioning, 3-8
- double precision, 12-6
- dummy argument name, 15-2
- element, 3-7, 3-9
- single precision, 12-6
- storage, B-9

ASCII character, 2-1

- Code Set, A-1

ASSIGN statement, 8-3

Assigned GO TO, 9-2

Assignment of .FALSE Value, 4-4

Assignment of .TRUE Value, 4-4

Assignment statements,

- arithmetic, 8-1
- ASSIGN, 8-1, 8-3
- logical, 8-1, 8-3

B

BACKFILE statement, 14-3

BACKSPACE statement, 14-2

Base/exponent operand types, 4-4

Basic external functions, 15-6

Table of, 15-8, 15-9

Blank, Line type, 2-4, 2-6

Blank common, 6-5

BLOCK data statement, 16-1

Block data subprograms, 16-1

Boldface italic type, 1-1

C

Calculation of DO loop iterations, 9-5

CALL statement, 15-9, 15-10

Categories of statements, 1-2

Character transfer,

maximum alphanumeric, 13-10

Character,

variable type by initial, 3-7

Characters

apostrophe representation, 13-11
ASCII, 2-1, A-1
continuation field, 2-3
digits, 2-2
nonprinting, 2-2
print control, 13-14
symbolic, 2-2
upper/lower case, 2-1

CLOSE Statement, 12-1, 12-2

Closing parenthesis, FORMAT statement, 13-12

COBOL-10, interaction with, B-29

Code Set,

ASCII Character, A-1

Codes

Table of conversion, 13-3
Table of numeric fields, 13-6

Comma delimiter, format specification, 13-11

Comment,

line identifier, 2-5
line type, 2-4
within a line, 2-5

Common,

blank, 6-5
labeled, 6-5

COMMON statement, 6-5, 6-6, B-9

Compilation control statements, 5-1

END statement, 5-1

Complex constant, 3-4

Complex data, 3-1

Complex quantities, transfer of, 13-6

Computation of DO loop iterations, 9-5, B-12

Computed GO TO, 9-2

Constants, 3-1

complex, 3-4
double octal, 3-4
double precision, 3-3
literal, 3-5
logical, 3-5
octal, 3-4
statement label, 8-3

Constant size,

double octal, 3-4
double precision, 3-3
integer, 3-2
octal, 3-4
real, 3-2

Continuation field, 2-3

Continuation lines, 2-4

CONTINUE statement, 9-9

Control characters for printer, 13-14

Control statements,

  device, 14-1, 14-3
  program, 9-1

Conversion,

  H, 13-10
  Result of literal, 13-11

Conversion codes, 13-3

Conversion for

  double precision data, 13-8
  mixed mode assignments, 8-2
  real data, 13-8

D

Data conversion, 13-8

DATA statement, 7-1, 7-2

Data statement label, 3-1

Data subprograms, BLOCK, 16-1

Data types, 3-1

Debug Line, 2-4, 2-6

Debugging FORTRAN-10 programs, D-1

Declarators,

  Array, 3-8
  type, 6-3

Definition of,

  array subscripts, 3-7
  external function, 15-5
  intrinsic function, 15-3
  statement function, 15-3

Delimiter,

  format specification comma, 13-11
  record, 13-1, 13-12

Descriptors,

  A (alphanumeric field), 13-9
  Action of Field, 13-5
  Field, 13-1, 13-2
  L (logical) field, 13-4, 13-9
  Literal Field, 13-10, 13-11
  numeric field, 13-4
  single quotes, 13-10
  record formatting field, 13-13
  T field, 13-13
  X field, 13-13

Descriptors and variables, interaction of, 13-6

Device OPEN/CLOSE statement option, 12-2

Device control statements, 14-1

  summary, 14-3

Dialog OPEN/CLOSE statement option, 12-8

Digit characters, 2-2

Dimension declaration, 3-9

DIMENSION statement, 6-1

Dimensioning of arrays, 3-8

  in COMMON, 6-6

Dimensions, adjustable, 6-2, 6-4

Directory, OPEN/CLOSE statement option, 12-5

DIRECTORY specification,

double precision arrays, 12-6
single precision arrays, 12-6

DO Loop, 9-5, 9-6

DO statement, 9-5

computations of iterations, 9-5, B-12
extended range, 4-7, 9-7
index variable, 9-5
nested, 9-6, 9-7
parameters, 9-5
transfer operations, 9-8
using floating point, B-12

Double octal constant, 3-4

Double octal data, 3-1

Double precision constant, 3-3

Double precision data conversion, 13-8

Dummy arguments, 15-1, 15-2

E

E notation, 3-3

Elements,

array, 3-7
language set, 1-1
order of array, 3-9

END FILE statement, 14-1, 14-2

END statement, 5-1

Entry Points,

multiple subprogram, 15-13
subroutine subprograms, 15-7

ENTRY statement, 15-13, 15-14, B-11

EQUIVALENCE statement, 6-1, 6-6, 6-7, B-10

Error reporting, B-8

Evaluation of expressions, 4-8

mixed mode, 4-9
nested subexpressions, 4-8

Executable statements, 1-1

Execution of RETURN statement, 15-12

Expressions,

arithmetic, 4-1, 4-2
complex arithmetic, 4-1
compound, 4-1
evaluation of, 4-8
evaluation of mixed mode, 4-4
logical, 4-2
mixed mode, 4-10
relational, 4-6
use of logical operands, 4-10

Extended range DO statement, 9-7

External FUNCTION statement, 15-5

External FUNCTION subprograms, 15-6, 15-12

External functions,

basic, 15-6, 15-8, 15-9
definitions of, 15-5
Octal arguments for, 15-12

External procedures, 15-1

EXTERNAL statement, 6-1, 6-7, 15-1, 15-3, 15-5

declaring function names, 6-8

F

Factors, scale, 13-7, 13-8

.FALSE. Value, assignment of, 4-4

Field codes, Table of numeric, 13-6

Field descriptors, A (alphanumeric), 13-9, 13-11

- action of, 13-5
- Alphanumeric, 13-9, 13-11
- Forms of, 13-2
- L (logical), 13-9
- literal, 13-10, 13-11
- numeric, 13-4
- Record formatting, 13-13
- Repeat format of, 13-2
- T, 13-13
- X, 13-13

Field width, variable numeric, 13-9

Fields,

- line continuation, 2-3, 2-4
- line statement, 2-3
- mixed numeric and alphanumeric, 13-11
- scale factors in, 13-7
- statement label, 2-3

File control statements, 12-1

Floating point DO loops, B-12

FORDDT

- commands, D-1
- loading and starting, D-1
- using for debugging, D-1

Form of

- BACKFILE statement, 14-3
- BACKSPACE statement, 14-2
- BLOCK data statement, 16-1
- CALL statement, 15-9
- END FILE statement, 14-2
- ENTRY statement, 15-13
- External FUNCTION statement, 15-5
- RETURN, Multiple Return, 15-10
- RETURN statement, 15-10
- REWIND statement, 14-2
- SKIP RECORD statement, 14-3
- statement functions, 15-3
- SUBROUTINE statement, 15-7
- UNLOAD statement, 14-2

Format specification comma delimiter, 13-11

FORMAT statement, 13-1

- closing parenthesis, 13-12
- READ/WRITE transfer to/from, 13-10

Format-Controller I/O statement processing, 13-6

Formatting field descriptors, 13-2

FOROTS

- ASCII data files, C-4
- binary data files, C-5
- calling sequences, C-6, C-12, C-13
- core requirements, C-23
- device assignments, C-24
- entry points, C-6
- error processing, C-3
- features of, C-1
- hardware requirements, C-1
- image binary files, C-5
- input/output facilities, C-3
- MACRO calls for FOROTS functions, C-7
- mixed mode data files, C-5
- software requirements, C-1
- source files, C-20

FORTRAN Subroutines, 15-10

FORTRAN-10

- global optimizer, B-15
- running the compiler, B-1
- switches, B-1
- writing programs, B-8

FUNCTION dummy arguments, 15-2

FUNCTION statement, 15-5

FUNCTION subprogram, 15-6

- names, 15-12

Functions, 15-1

- basic external, 15-6, 15-8, 15-9
- dummy arguments in, 15-2
- external, 15-1, 15-5
- generic names for, 15-6, 15-7
- intrinsic, 15-1, 15-3, 15-4, 15-5

logical, 4-6
Statement, 15-1, 15-3
use of library name for, 15-5

G

G numeric conversion code, 13-7

Generic names for functions, 15-6, 15-7

Global optimizer

constant folding and propagation, B-18
elimination of common subexpressions, B-16
global register allocation, B-19
improper function references, B-19
optimization techniques, B-16
programming techniques for effective optimization, B-19
reduction of operator strength, B-17
removal of constant computation from loops, B-17
removal of inaccessible code, B-19

GO TO Statement, 9-1, 9-2, 9-3

assigned, 9-2, 9-3
computed, 9-2
types of, 9-1
unconditional, 9-2

H

H Conversion, 13-10

Hierarchy, of operators, 4-8, 4-9

Hollerith literal, 3-5

I

I/O statements processing, 13-6

Identifier,

array elements, 3-9
comment line, 2-5

IF STATEMENT, 9-3, 9-4

arithmetic, 9-3
logical, 9-4
logical two-branch, 9-4

IMPLICIT statement, 6-4, 6-5

Increment parameter DO statement, 9-5

Initial character, typing variables by, 3-7

Initial line, 2-4

statement number, 2-3, 2-4
use of tab, 2-3

Initial parameters DO statement, 9-5

Initial tab, 2-2, 2-3, 2-4

Input, line-sequenced, 2-6

Input transfers,

NAMELIST controlled, 11-2

Integer constants

size, 3-2

Integer data, 3-1

Integer variable types, 3-6

Interaction of descriptors and variables, 13-6

Interfacing with non-FORTRAN-10 programs and files, B-19

Internal procedures, 15-1

Intrinsic functions, 15-3, 15-4, 15-5

Iterations, calculation of DO loop, 9-5

L

L (logical) field descriptor, 13-9

Label statement, 3-5, 15-11

Label in data statement, 7-1

Label dummy arguments, 15-2

Label field in statement, skipping, 2-3

Label in CALL statement, 15-10

Labeled common area, 6-5

Language set, elements of, 1-1

Library subroutines, 15-15 through 15-19

Line Identifier for comments, 2-5

Line Printer control characters, 13-14

Line-sequenced Input, 2-6

Line statement field, 2-3

Line Types, 2-4

Literal constant, 3-5

  in CALL statements, 15-10

Literal conversion, 13-11

Literal data, 3-1, 3-5

Literal field description, 13-10, 13-11

Literals, Hollerith, 3-5

Logical

  assignment statement, 8-3
  bit combinations, 4-6
  constant, 3-5
  data, 3-1, 3-5
  expression form, 4-4
  expressions, 4-3
  field descriptor, 13-9
  functions, 4-6
  IF statement, 9-4
  operations binary truth table, 4-6
  operations truth table, 4-5
  operators, 4-4
  two-branch IF statement, 9-4
  variable types, 3-6

Lower case characters, 2-1

M

MACRO-10, interaction with, B-24

Mixed mode

  assignment, rules for conversion, 8-2
  expression, 4-9
  expression, evaluation of, 4-10
  expression, use of logical operand, 4-10

Mixed numeric and alphanumeric fields, 13-11

Monitor commands, B-2

Multi-statement line, 2-5

Multiple record specification, 13-12

Multiple returns,

  definitions, 15-10
  RETURN statement with, 15-10

Multiple subprogram entry points, 15-13

N

Name, symbolic, 3-5, 3-6

NAMELIST controlled I/O transfers, 11-2, 11-3

NAMELIST statement, 11-1, 11-2

Names,

  Function generic, 15-6, 15-7
  FUNCTION Subprogram, 15-12

Nested DO statements, 9-6, 9-7

Nested subexpressions, 4-8

Nonexecutable statements, 1-1

Numbers for statement lines, 2-3, 2-4

Number of RETURNS, 15-11

Numeric and alphanumeric fields, mixed, 13-11

Numeric Conversion code, G, 13-7

Numeric field codes, 13-6

Numeric field descriptors, 13-4

Numeric field width, variable, 13-9

Numeric fields with scale factors, 13-7

O

Octal constants, 3-4

Octal data, 3-1, 3-4

OPEN/CLOSE statement Options

- access, 12-2
- associate variable, 12-8
- buffer count, 12-7
- density, 12-8
- dialogue, 12-8
- directory, 12-5
- dispose, 12-4
- mode, 12-3
- parity, 12-8
- protection, 12-5
- record size, 12-7
- summary, 12-9
- unit, 12-2
- version, 12-7

OPEN statement, 12-1, 12-2

Operand types, 4-1

Operation

- of DO loop, 9-6
- of DO statement transfer, 9-7

Operator hierarchy, 4-8, 4-9

Operators,

- arithmetic, 4-1
- logical, 4-4
- relational, 4-7

Options, summary of OPEN/CLOSE statement, 12-9

Output transfers, NAMELIST controlled, 11-3

P

P Scale Factor, 13-8

Parameters of DO statement, 9-5

Parenthesis in FORMAT statement, 13-12

Parenthesized subexpressions, 4-8

PAUSE statement, 9-10

Printer control characters, 13-14

Procedures (functions), 15-1

Programs, source, 1-1

Protection option for OPEN/CLOSE statement, 12-5

Q

Quotes descriptor, 13-13

R

RAN 15-9
SAVRAN 15-18
SETRAN 15-18

Range of DO loop, 9-5

READ transfer, formatted, 13-10

Reading a FORTRAN-10 listing, B-3

Real

variables, 3-6, 3-7
data, 3-1, 3-2
data, conversion of, 13-8
constant, size of, 3-2, 3-3

Record delimiter, 13-1, 13-12

Record formatting field descriptors, 13-13

Record specification, multiple, 13-12

Referencing external FUNCTION subprograms, 15-12

Relational expressions, 4-6, 4-7

Remarks, 2-3, 2-5, 2-6

Reordering of computations, B-14

Repeat format of field descriptors, 13-2

Replacement of dummy arguments, 15-2

Representing apostrophe characters, 13-11

Result of literal conversion, 13-11

Result of statement function, 15-3

RETURN statement, 15-10, 15-11, 15-12

Subprogram, 15-7

Returns, multiple, 15-10, 15-11

REWIND statement, 14-2

Rules for FUNCTION Subprogram, 15-6

Rules for multi-statement line, 2-5

Rules for Use of ENTRY statement, 15-13, 15-14

Rules, form and use of dummy arguments, 15-2

Rules for SUBROUTINE Statement, 15-9

Running the FORTRAN-10 compiler, B-1

S

Scale Factors, 13-7, 13-8

Single quotes descriptor, 13-10

Size of

double octal constant, 3-5
double precision constant, 3-3
Integer constant, 3-2
Octal constant, 3-4
Real constant, 3-2

SKIPFILE statement, 14-3

SKIPRECORD Statement, 14-3

Skipping label field, 2-3

Slash (/) used as record delimiter, 13-1, 13-12

Source programs, 1-1

Specification of multiple record, 13-12

Specification comma delimiter, 13-11

Specification statements, 6-1

Statement,

Actual Arguments for CALL, 15-9
Arguments for ENTRY, 15-13
Arithmetic assignment, 8-1

Statements,

ACCEPT, 10-15, 10-16
ASSIGN, 8-3
BACKFILE, 14-1, 14-3
BACKSPACE, 14-1, 14-2
BLOCK data, 16-1
CALL, 15-9
CLOSE, 12-1
COMMON, 3-9, 6-1, 6-5
CONTINUE, 9-1, 9-9
DATA, 7-1, 7-2, 7-3
DECODE, 10-19, 10-20, 10-21
DIMENSION, 3-9, 6-1

DO, 9-1, 9-5
ENCODE, 10-19, 10-20, 10-21
END, 5-1
ENDFILE, 14-1
ENTRY, 15-13
EQUIVALENCE, 6-1, 6-6
EXTERNAL, 6-1, 6-7, 15-3, 15-5
FIND, 10-18
FORMAT, 13-1 through 13-14
FUNCTION, 15-5
GO TO, 9-1
IF, 9-1, 9-3
IMPLICIT, 6-1, 6-4
NAMELIST, 11-1 through 11-3
OPEN/CLOSE, 12-1 through 12-9
PAUSE, 9-10
PRINT, 10-16
PUNCH, 10-17
READ, 10-9 through 10-12
REREAD, 10-12, 10-13
RETURN, 15-10
REWIND, 14-2
SKIPFILE, 14-3
SKIPRECORD, 14-3
STOP, 9-9
SUBROUTINE, 15-7
TYPE, 10-18
UNLOAD, 14-2
WRITE, 10-13 through 10-15

Storage of arrays, B-9

Subexpressions, 4-8

Subprogram names, FUNCTION, 15-12

Subprogram RETURN statement, 15-7

Subprograms,

block data, 16-1
multiple entry points for, 15-13
Referencing External FUNCTION, 15-12
Subroutine, 15-7

SUBROUTINE Statement, 15-7

Subroutines,

FORTRAN, 15-10
library, 15-15 through 15-19
programming considerations, B-14

Subscripts, definition of array, 3-7

Summary of device control statements, 14-3

Summary of OPEN/CLOSE statement Options, 12-9

Switches available with FORTRAN-10 compiler, B-1

Symbolic

characters, 2-2
name, 3-5, 3-6
relational operators, 4-7

T

T Field Descriptor, 13-13

T (TRACE) option, 9-10, 9-11

Tab, use of in initial line, 24

Tables

basic external functions, 15-8, 15-9
conversion codes, 13-3
intrinsic functions, 15-4, 15-5
library functions, 15-4, 15-5
library subroutines, 15-15, 15-16, 15-17, 15-18, 15-19
numeric field codes, 13-6
print control characters, 13-14

Teletype printer control characters, 13-14

Terminal Parameter DO statement, 9-5

TRACE option, 9-10, 9-11

TRACE routine, 9-10, 9-11

Transfer of COMPLEX quantities, 13-6

Transfer operation, DO statement, 9-8

Transfer with FORMAT statement, 13-10

.TRUE. value, assignment of, 4-4

Type declarators, 6-3

Type of External FUNCTION Statement, 15-5

Type of statement function, 15-3

Type Specification statements, 6-1, 6-3

Types of dummy arguments, 15-2

U

Unconditional GO TO, 9-2

Unit option in OPEN/CLOSE statement, 12-2

UNLOAD statement, 14-2

Unspecified scale factor, 13-8

Upper case characters, 2-1

Use of

- COMMON, B-9
- EQUIVALENCE statements, B-10
- ENTRY statements, B-11
- Floating point DO loops, B-12

Using library name for user function, 15-5

V

Variables,

- complex, 3-6
- DO index, 9-5
- double precision, 3-6
- dummy argument, 15-2
- integer, 3-6
- logical, 3-6
- numeric field width, 13-9
- real, 3-6
- types of, 3-6
- types of initial characters, 3-7

W

Width of variable numeric field, 13-9

WRITE transfer from FORMAT statement, 13-10

Writing FORTRAN-10 Programs for execution on non-DEC machines, B-8

Writing user programs, B-1

X

X field descriptor, 13-13

DECsystem-10
FORTRAN-10 Language Manual
DEC-10-LFORA-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

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